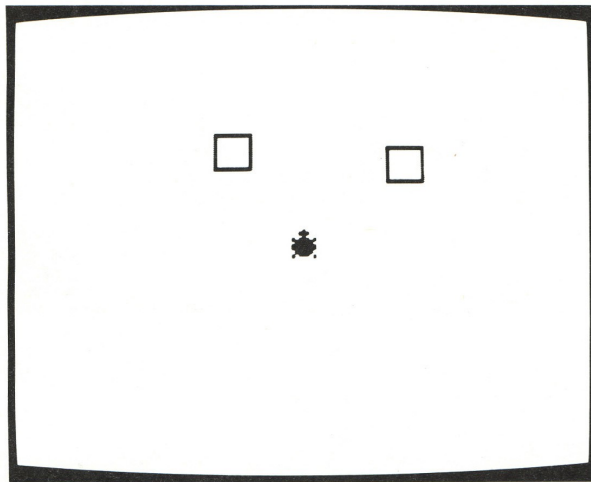


3

Games

Boxgame

When you type `BOXGAME`, two square boxes are put on the screen. They are the targets. A turtle appears in the center of the screen. The goal of the game is to put the turtle inside each box. After you put the turtle inside a box, the box vanishes. `BOXGAME` gives you experience moving and turning the turtle.



After `BOXGAME` sets up the boxes and the turtle, it activates demons to watch for the turtle crossing over the lines of the boxes. Then the procedure stops, and you take over and control the turtle directly using commands like `FD`, `BK`, `LT`, or `RT`. When one of the turtles collides with a line, a demon invokes instructions that make the box disappear.

`BOXGAME` can be modified so that you control the turtle in different ways. You might want to use special commands, like `F` for `FD 10` and `R` for `RT 15`. You might prefer to use a joystick to control the turtle.

In the following discussion I begin by showing how `BOXGAME` was constructed so that you use Logo primitives like `FD` and `RT`. I also show how to introduce new commands like `F` and `R` to the game. Lisa Delpit then describes her version of `BOXGAME`, which she made for some young children

GAMES

she was working with. I later describe how to change BOXGAME so that you control the turtle with a joystick in port 1. Then I add more frills to the joystick version.

The Procedures

BOXGAME sets up turtle 0 and then turns the rest of the job over to SETUP. The player is in direct control of the turtle.

```
TO BOXGAME
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP
PR [NOW PUT THE TURTLE IN EACH BOX]
END
```

SETUP calls SETBOXES to put the two targets on the screen and then alerts the demons to watch for turtle 0 crossing over lines drawn by pens 0 or 1. When either event happens, the pen color is changed to the background color and thus the box becomes invisible.

```
TO SETUP
CS PU
SETBOXES
WHEN OVER 0 0 [SETPC 0 BG]
WHEN OVER 0 1 [SETPC 1 BG]
SETPN 2 SETPC 2 24
END
```

The boxes are drawn in different colors. Their positions are chosen at random and are likely to be different each time the game is played.

```
TO SETBOXES
SETPN 0 SETPC 0 7
DRAWBOX RANDOM 90
SETPN 1 SETPC 1 100
DRAWBOX 0 - RANDOM 90
END
```

SETBOXES uses DRAWBOX to put a box on the screen.

```
TO DRAWBOX :ANGLE
PU SETH :ANGLE
FD RANDOM 100
SETH 0 PD
BOX 20
PU HOME
END
```

```
TO BOX :SIDE
REPEAT 4 [FD :SIDE RT 90]
END
```

This kind of game can be fun for a while. But it can also be hard work for very young children! Thus you might want to add procedures that will

let the user type in single-key commands for controlling the turtle. For example, when the user types F, the turtle moves forward twenty units.

TO F	TO B	TO L	TO R
FD 20	BK 20	LT 15	RT 15
END	END	END	END

Bye-Bye Boxes

(A Modification of Boxgame)

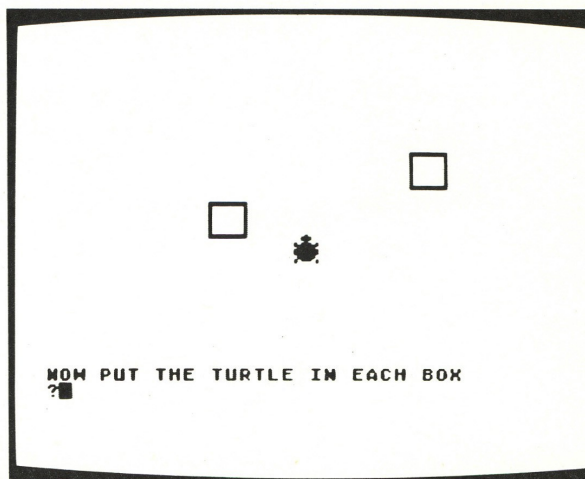
I used Cynthia's BOXGAME with a group of five-year-olds to help them in their left-right orientation, and they loved it. But while they improved their ability to direct the turtle when the turtle's direction was at HEADING 0 (that is, when the turtle's left and right were the same as their left and right), they were still thoroughly confused when the turtle was headed in any other direction. To help solve this problem I modified BOXGAME so that the squares appear in any of eight directions (0, 45, 90, 135, 180, 235, 270, or 315) on the screen at different distances from the center. I also set the turtle up in the center of the screen, but now facing in one of the eight directions. I added sound effects too, partially because I thought the kids would find it interesting but mostly because I enjoy playing with TOOT. The children came up with the catchy name.

The procedures are almost the same as those for BOXGAME, with the additions of GETTURN, which generates the number for the turtle's heading, and DEC.SOUND and INC.SOUND, which add the sound effects.

The Procedures

I will point out where I made changes to BOXGAME. BYEBYE is like BOXGAME except that it calls SETUP.BYE.

```
TO BYEBYE
  TELL [0 1 2 3] HT
  TELL 0 SETSH 0 ST
  SETUP.BYE
  PR [NOW PUT THE TURTLE IN EACH BOX]
  END
```



GAMES

In SETUP.BYE, I add sound to the instructions for the demons.

```
TO SETUP.BYE
CS PU
SETBOXES.BYE
WHEN OVER 0 0 [SETPC 0 BG DEC.SOUND 3500]
WHEN OVER 0 1 [SETPC 1 BG DEC.SOUND 3500]
SETPN 2 SETPC 2 24
END
```

I changed the colors of the squares in SETBOXES.BYE.

```
TO SETBOXES.BYE
SETPN 0 SETPC 0 7
DRAWBOX.BYE
SETPN 1 SETPC 1 100
DRAWBOX.BYE
END
```

DRAWBOX.BYE sets up the turtle for drawing each box at an angle that is a multiple of 45 and at a distance of 25 to 70 steps from the center. This distance is not far enough away to be hidden behind the text at the bottom of the screen. DRAWBOX.BYE, then, turns the turtle to a heading that is a multiple of 45, and the game begins.

```
TO DRAWBOX.BYE
PU
SETH GETTURN
FD 25 + RANDOM 45
SETH 0 PD
INC.SOUND 3000
BOX 20
PU HOME
END
```

I wrote GETTURN so that it outputs one of eight possible numbers, all multiples of 45.

```
TO GETTURN
OP 45 * RANDOM 8
END
```

A sound of increasing frequency accompanies the drawing of the box. A sound of decreasing frequency accompanies the disappearance of the box.

```
TO INC.SOUND :FRE
IF :FRE > 3500 [TOOT 1 4000 8 3 STOP]
TOOT 1 :FRE 6 3
INC.SOUND :FRE + 100
END
```

```
TO DEC.SOUND :FRE
IF :FRE < 3000 [TOOT 1 1000 8 3 STOP]
TOOT 1 :FRE 6 3
DEC.SOUND :FRE - 100
END
```

The following three procedures are unchanged.

```
TO BOX :X
REPEAT 4 [FD :X RT 90]
END
```

```
TO F
FD 20
END
```

```
TO B
BK 20
END
```

The next procedures are changed so that the turtle turns 45 degrees.

```
TO R
RT 45
END
```

```
TO L
LT 45
END
```

Back to Boxgame

Using a Joystick

Another variation of this game is to attach a joystick to the computer and use the joystick to control the turtle. In the next example, pressing the joystick button moves the turtle forward five steps. The joystick moved to the left turns the turtle left 15 degrees; the joystick moved to the right turns the turtle right 15 degrees. To do this BOXGAME has to be changed and a couple of new procedures have to be written for the joystick.

```
TO JOYGAME
CT SS
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP
PR [THE JOYSTICK TURNS THE TURTLE]
PR [THE BUTTON MOVES THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD
END
```

```
TO JOYREAD
IF JOYB 0 [FD 5 ]
IF EQUALP PC 0 PC 1 [STOP]
JOYRD JOY 0
JOYREAD
END
```


GAMES

```

TO JOYRD :STICK
IF :STICK = 6 [LT 15]
IF :STICK = 2 [RT 15]
END

```

Extending JOYGAME

JOYGAME will set the turtle's speed. I use a speed of 100, but you might want to change this. This time when the turtle goes over a pen line, the background changes color. Finally, the game starts up again.

JOYGAME needs to be changed. I also want to change the setting-up procedure. Let's rename the new versions of these procedures.

```

TO NEWGAME
CT SS
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETJOY
SETSP 100
PR [THE JOYSTICK TURNS THE TURTLE]
PR [THE BUTTON MOVES THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD
NEWGAME
END

TO SETJOY
CS PU
SETBOXES
WHEN OVER 0 0 [FLASH BG SETPC 0 BG]
WHEN OVER 0 1 [FLASH BG SETPC 1 BG]
SETPN 2 SETPC 2 24
END

```

Here is the new procedure.

```

TO FLASH :BG
REPEAT 6 [SETBG RANDOM 100 WAIT 5]
SETBG :BG
END

```

PROGRAM LISTING

```

TO BOXGAME
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP
PR [NOW PUT THE TURTLE IN EACH BOX]
END

TO SETUP
CS PU

```

```

SETBOXES
WHEN OVER 0 0 [SETPC 0 BG]
WHEN OVER 0 1 [SETPC 1 BG]
SETPN 2 SETPC 2 24
END

TO BOX :SIDE
REPEAT 4 [FD :SIDE RT 90]
END

```

```

TO SETBOXES
  SETPN 0 SETPC 0 7
  DRAWBOX RANDOM 90
  SETPN 1 SETPC 1 100
  DRAWBOX 0 - RANDOM 90
END

```

```

TO DRAWBOX :ANGLE
  PU SETH :ANGLE
  FD RANDOM 100
  SETH 0 PD
  BOX 20
  PU HOME
END

```

```

TO BYEBYE
  TELL [0 1 2 3] HT
  TELL 0 SETSH 0 ST
  SETUP.BYE
  PR [NOW PUT THE TURTLE IN EACH BOX]
END

```

```

TO SETUP.BYE
  CS PU
  SETBOXES
  WHEN OVER 0 0 [SETPC 0 BG DEC.SOUND ►
    3500]
  WHEN OVER 0 1 [SETPC 1 BG DEC.SOUND ►
    3500]
  SETPN 2 SETPC 2 24
END

```

```

TO SETBOXES.BYE
  SETPN 0 SETPC 0 7
  DRAWBOX.BYE
  SETPN 1 SETPC 1 100
  DRAWBOX.BYE
END)

```

```

TO DRAWBOX.BYE
  PU
  SETH GETTURN
  FD 25 + RANDOM 45
  SETH 0 PD
  INC.SOUND 3000
  BOX 20
  PU HOME
END

```

```

TO GETTURN
  OP 45 * RANDOM 8
END

```

```

TO INC.SOUND :FRE

```

```

  IF :FRE > 3500 [TOOT 1 4000 8 3 STOP]
  TOOT 1 :FRE 6 3
  INC.SOUND :FRE + 100
END

```

```

TO DEC.SOUND :FRE
  IF :FRE < 3000 [TOOT 1 1000 8 3 STOP]
  TOOT 1 :FRE 6 3
  DEC.SOUND :FRE - 100
END

```

```

TO F
  FD 20
END

```

```

TO B
  BK 20
END

```

```

TO R
  RT 45
END

```

```

TO L
  LT 45
END

```

```

TO JOYGAME
  CT SS
  TELL [0 1 2 3] HT
  TELL 0 SETSH 0 ST
  SETUP
  PR [THE JOYSTICK TURNS THE TURTLE]
  PR [THE BUTTON MOVES THE TURTLE]
  PR [NOW PUT THE TURTLE IN EACH BOX]
  JOYREAD
END

```

```

TO JOYREAD
  IF JOYB 0 [FD 5 ]
  IF EQUALP PC 0 PC 1 [STOP]
  JOYRD JOY 0
  JOYREAD
END

```

```

TO JOYRD :STICK
  IF :STICK = 6 [LT 15]
  IF :STICK = 2 [RT 15]
END

```

```

TO NEWGAME
  CT SS
  TELL [0 1 2 3] HT
  TELL 0 SETSH 0 ST

```

```

SETJOY
SETSP 100
PR [THE JOYSTICK TURNS THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD.NEW
NEWGAME
END

```

```

TO JOYREAD.NEW
IF EQUALP PC 0 PC 1 [STOP]
JOYRD JOY 0
JOYREAD.NEW
END

```

```

TO SETJOY
CS PU
SETBOXES
WHEN OVER 0 0 [FLASH BG SETPC 0 BG]
WHEN OVER 0 1 [FLASH BG SETPC 1 BG]
SETPN 2 SETPC 2 24
END

```

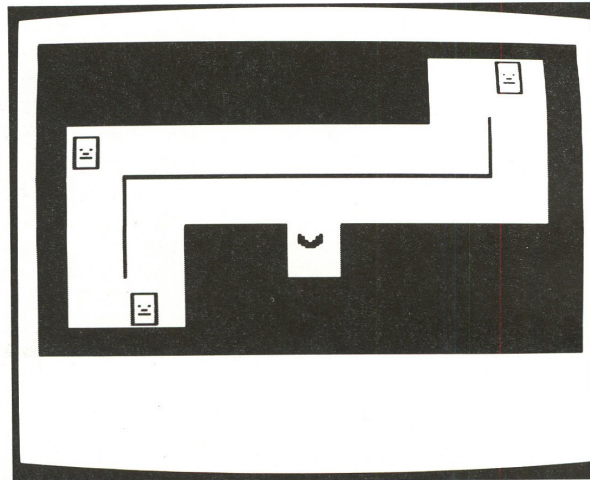
```

TO FLASH :BG
REPEAT 6 [SETBG RANDOM 100 WAIT 5]
SETBG :BG
END

```

Pacgame

PACGAME was inspired by PAC-MAN™ and designed as a learning tool for beginners. You play the game by using turtle commands to move the gobbling pacman around the game board. The game's special effects and features are activated entirely by demons. Thus a player types all commands directly to Logo and demons take care of the game's actions.



Here are the rules I decided on for PACGAME.

- The game is played on a game board. There is a pacman and three targets. Unlike PAC-MAN's ghosts, the targets in PACGAME are stationary.
- Once play begins, it continues until all three targets are gobbled.
- Each target is worth 10,000 points and explodes when it is gobbled.
- All turns need to be multiples of 90 degrees because the pacman can gobble in only four directions. If any other turn is made, the pacman will change back into a turtle and complain.

PAC-MAN is a trademark of Bally Midway Manufacturing Company

By Michael Grandfield.

In addition to the rules, the pacman bounces back onto the game board whenever it goes out of bounds.

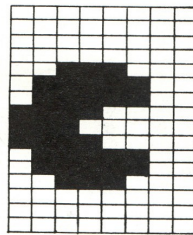
I'd like to present PACGAME by showing the playing pieces and game board I designed and talking about the demons that bring the game to life.

The Playing Pieces and the Game Board

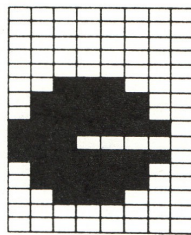
Let's look at these one at a time.

The Pacman

I wanted to make a pacman that could gobble and also behave like a turtle. It took two shapes to animate the pacman, one for an open mouth and one for a closed mouth.



: SHAPE 3



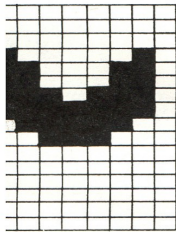
: SHAPE 4

Animating the pacman without limiting its turtle capabilities or interfering with typing in a new command was a problem. However, I found an effective solution by using the once-per-second demon. Here it is.

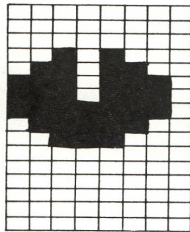
```
WHEN 7 [SETSH 1 WAIT 30 SETSH 2]
```

This demon runs the instruction list [SETSH 1 WAIT 30 SETSH 2] once every second. This makes the animation continue steadily until the demon is halted.

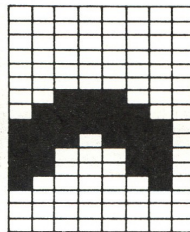
Next I decided that the pacman should be able to gobble in four different directions (up, down, right, and left) and made six more pacman shapes.



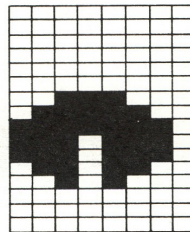
: SHAPE 1



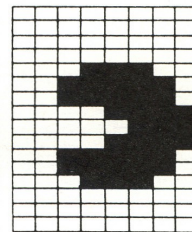
: SHAPE 2



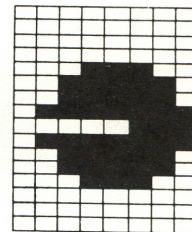
: SHAPE 5



: SHAPE 6



: SHAPE 7



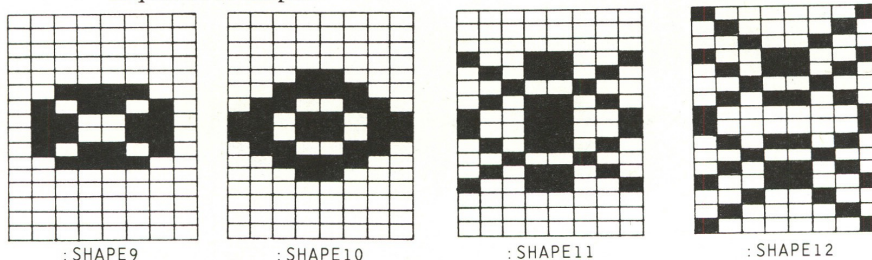
: SHAPE 8

Later I explain how the pacman chooses the pair of shapes that corresponds to its heading.

The Targets

Next I designed a target. The design I settled on has a distinct outline to make it easy to see how far the pacman is from hitting it. It also has a sinister face.

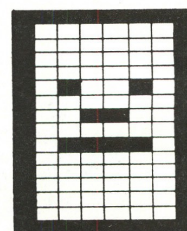
I wanted this target to explode as it was being gobbled, so I made this sequence of shapes.



In the game, three turtles are targets. Collision-detection demons tell when a target has been hit. Here's an example.

WHEN TOUCHING 0 1 [EXPLODE 1]

:SHAPE13

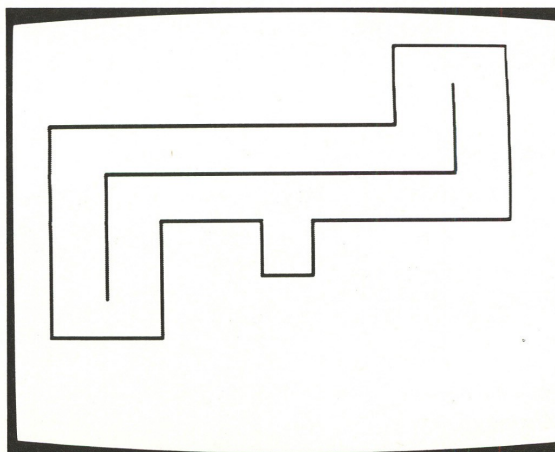


EXPLODE gives the turtle representing the target each of the explosion shapes in quick succession and also changes the turtle's color with each change of shape.

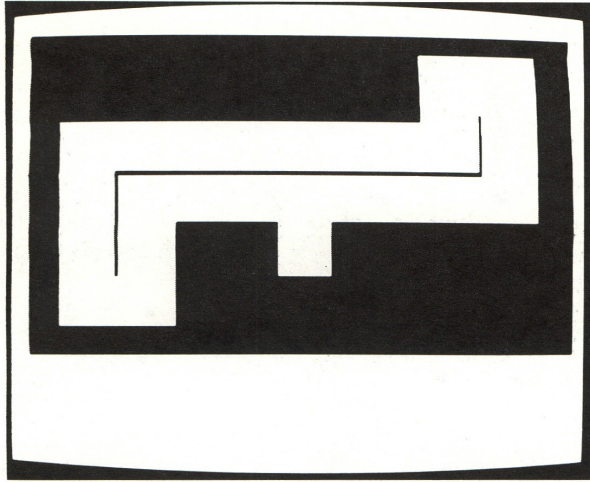
Later I added new instructions for these demons, in order to score the game and congratulate the player.

The Game Board

Originally the game board was a thin outline. It looked like this.



There was a problem with this approach. I wanted a demon to detect any collision between the pacman and a border of the game board. Also, I wanted to keep the pacman in bounds by having it bounce back from a collision. I discovered that these borders were too thin. The demon often



failed to detect a collision. My solution was to make the game board have very thick borders.

Here is the procedure that draws the game board. Notice that I used two pens of different colors to draw the board.

```
TO DRAW.BOARD
  SETPC 0 22
  SETPC 1 26
  TELL 0 SETPN 0 PU HT
  SETPOS [-160 120] PD SETH 90
  REPEAT 10 [FD 320 SETY YCOR - 1]
  REPEAT 40 [FD 230 SETPN 1 FD 70 SETPN 0 FD 20 SETY YCOR - 1]
  REPEAT 60 [FD 20 SETPN 1 FD 280 SETPN 0 FD 20 SETY YCOR - 1]
  FD 20 SETPN 1 FD 280 SETPN 0 FD 20
  REPEAT 34 [FD 20 SETPN 1 FD 70 SETPN 0 FD 58 SETPN 1 FD 30
    SETPN 0 FD 142 SETY YCOR - 1]
  REPEAT 30 [FD 20 SETPN 1 FD 70 SETPN 0 FD 230 SETY YCOR - 1]
  REPEAT 15 [FD 320 SETY YCOR - 1]
  PU
  SETPOS [-105 -20] SETH 0 PD
  FD 60 RT 90 FD 210 LT 90 FD 35
  PU
  END
```

Bringing the Game to Life

The main procedure is PACGAME. It calls SETUP and PLAY.

```
TO PACGAME
  SETUP
  PLAY
  END
```

Setting Up

SETUP clears all graphics and text from the screen, and calls PUT.SHAPES and DRAW.BOARD.

GAMES

```

TO SETUP
CS SS CT
PUT.SHAPES 1
SETUP.VARS
DRAW.BOARD
END

```

PUT.SHAPES puts all the shapes, which are stored as lists of numbers, into the shape slots. It also erases the variables that contained the lists. (I guess I always like to free up as much of the workspace as I can.)

```

TO PUT.SHAPES :NUM
IF :NUM > 15 [STOP]
PUTSH :NUM THING WORD "SHAPE :NUM
ERN WORD "SHAPE :NUM
PUT.SHAPES :NUM + 1
END

```

SETUP.VARS sets up variables that the program uses.

```

TO SETUP.VARS
MAKE "270 7
MAKE "180 5
MAKE "90 3
MAKE "0 1
MAKE "POS3 [-95 -40]
MAKE "POS2 [-130 55]
MAKE "POS1 [116 100]
MAKE "PURPLE 70
MAKE "PACMAN 0
MAKE "BLACK 0
MAKE "TARGET 9
END

```

You have already seen DRAW.BOARD in the description of the game pieces.

The Play

The procedure PLAY calls two setup procedures, SET.PIECES and SET.DEMONS. Once these procedures have been run, the game begins.

```

TO PLAY
SET.PIECES
SET.DEMONS
END

```

SET.PIECES sets the score to 0 and places the pacman and the targets on the gameboard in the correct positions to begin the game. It also creates the variable PAC.POSITION.

During the game PAC.POSITION is given the value of the pacman's current position. This value changes at regular intervals, provided that the pacman remains in bounds. If the pacman goes out of bounds

PAC.POSITION is not given a new value, so the pacman can bounce back to the position that is the value of PAC.POSITION.

```
TO SET.PIECES
MAKE "SCORE 0
TELL 0 HT HOME SETC 44
ASK [1 2 3] [HT PU SETC 7 SETSH :TARGET]
ASK [1 2 3] [EACH [SETPOS THING WORD "POS WHO] ST]
CT ST
MAKE "PAC.POSITION POS
END
```

SET.DEMONS creates demons. These demons animate the pacman and detect collisions. The procedures that these demons call are the guts of the game.

```
TO SET.DEMONS
WHEN 7 [IF MEMBERP HEADING [0 90 180 270]
      [ANIMATE.PACMAN]
      [REVEAL.TURTLE]]
WHEN OVER 0 0 [STAY.IN.BOUNDS]
WHEN TOUCHING 0 1 [BULLSEYE 1]
WHEN TOUCHING 0 2 [BULLSEYE 2]
WHEN TOUCHING 0 3 [BULLSEYE 3]
END
```

Procedures Called by the Demons

Animating the Pacman

You can see that I have changed the instructions for the once-per-second demon (7) from the earlier example. The demon checks to see if the pacman has heading 0, 90, 180, or 270. If so, it animates the pacman; otherwise it reveals the original turtle shape. As you will see, this demon is able to do several different jobs neatly.

The procedure ANIMATE.PACMAN uses two interesting programming tricks.

```
TO ANIMATE.PACMAN
SETSH THING HEADING
WAIT 10
UPDATE
SETSH 1 + THING HEADING
END
```

The first trick is that I have given each shape a variable name that is the same as the heading associated with the shape. For example, the shape that gobbles upward is used when the turtle has a heading of 0, and is given the name "0. The variable for this shape is created by the instruction

```
MAKE "0 1
```

GAMES

Now if I type

```
PR :0
```

or

```
PR THING 0
```

Logo will respond

```
1
```

If the pacman's heading is 0, I can also type

```
SETSH THING HEADING
```

and Logo will respond as if I had typed

```
SETSH 1
```

The second trick is that I decided to use the bit of time between shape changes to call the procedure `UPDATE`. This procedure checks to see if the pacman is still in bounds. If so, `UPDATE` gives a new value to the variable `PAC.POSITION`.

```
TO UPDATE
IF COND OVER 0 1 [MAKE "PAC.POSITION POS]
END
```

To sum up, `ANIMATE.PACMAN` changes the shape of the pacman to correspond to its heading, animates its gobbling, and calls `UPDATE`.

Here is the procedure `REVEAL.TURTLE`. It is called whenever a turn that is not a multiple of 90 degrees is made.

```
TO REVEAL.TURTLE
SETSH 0
SETCURSOR [0 19]
TYPE [\ FIX HEADING TO GET BACK IN THE GAME\ ]
END
```

`REVEAL.TURTLE` makes it clear when a turn is not within the rules by showing the turtle shape and protects the game from crashing by allowing the pacman to respond to any turning instruction.

Staying in Bounds

`STAY.IN.BOUNDS` is called whenever the pacman bumps into a boundary of the board. It makes a bumping sound, sends the pacman back to `:PAC.POSITION`, and complains `OUCH!!!`.

```
TO STAY.IN.BOUNDS
TOOT 0 100 7 15
TOOT 1 200 7 15
SETPOS :PAC.POSITION
PR [] PR "OUCH!!! PR []
END
```


Gobbling the Targets

BULLSEYE is called whenever the pacman hits a target. It temporarily stops the gobbling animation by halting the once-per-second demon and lets you know that the pacman has hit a target by changing the pacman's color. Next BULLSEYE updates the score and tells the gobbled target to explode. It also offers some congratulations and prints your score on the screen. Finally BULLSEYE checks the score to see whether to continue the game or declare a victory.

```

TO BULLSEYE :WHICH.TARGET
  WHEN 7 [] SETC :BLACK
  ADD.TO.SCORE
  EXPLODE :WHICH.TARGET
  ( PR :SCORE [TO BE EXACT] )
  WAIT 90
  IF NOT :SCORE = 30000 [ON.WITH.THE.GAME] [VICTORY]
  END

TO ADD.TO.SCORE
  MAKE "SCORE 10000 + :SCORE
  CT PR [BOINK!!! WOWIEE!!!]
  PR [SCORE LOTS OF POINTS FOR YOU!]
  PR []
  END

```

The procedure EXPLODE animates the explosion by calling EXPLODE1.

```

TO EXPLODE :TARGET
  ASK :TARGET [EXPLODE1 [9 10 11 12 13 12 13 12 13 12]
                    [22 31 55 79 55 79 55 79 55 7]
                    HT HOME]
  END

TO EXPLODE1 :SHAPES :COLORS
  IF EMPTY :SHAPES [STOP]
  TOOT 0 20 10 7 TOOT 1 25 10 6
  SETSH FIRST :SHAPES
  SETC FIRST :COLORS
  EXPLODE1 BF :SHAPES BF :COLORS
  END

```

The tricky part of EXPLODE1 was synchronizing sound and animation. I use both voices to emit a sound before each shape and color change. Thus I use two TOOT commands. The shape and color changes begin before the sound dies away, so all three events happen together.

On with the Game

ON.WITH.THE.GAME continues the game after a target has been gobbled. It resets the pacman's color and resets the once-per-second demon.

GAMES

```

TO ON.WITH.THE.GAME
TELL :PACMAN SETC :PURPLE
WHEN 7 [IF MEMBERP HEADING [0 90 180 270]
[ANIMATE.PACMAN]
[REVEAL.TURTLE]]

END

```

Winning the Game

VICTORY is called only when the score has reached 30,000. It celebrates winning the game by playing a rousing fanfare and flashing the background color in time to the music.

```

TO VICTORY
RECYCLE
TELL 0
CT
PR [THE WINNER, AND STILL CHAMP!]
FANFARE
WAIT 180
PR [DO YOU WANT TO PLAY AGAIN?]
END

```

FANFARE calls FANFARE1, which plays some rousing music to celebrate.

```

TO FANFARE
FANFARE1 55 110 39 40 7040
FANFARE1 35 70 31 48 7040
FANFARE1 30 60 23 56 7040
FANFARE1 25 50 15 64 7040
FANFARE1 30 50 78 71 7680
FANFARE1 120 200 63 56 7680
SETPC 0 22
SETPC 1 26
SETBG 64
SETENV 0 15 SETENV 1 15
TOOT 0 240 15 240 TOOT 1 400 15 240
END

```

FANFARE1 invokes T00T for both voices, then changes the color of the game board and the background. Each time FANFARE1 repeats, the notes increase an octave in pitch, until the :HIGHFREQ limit is reached.

```

TO FANFARE1 :FREQ0 :FREQ1 :PENCOLOR
:SCREENCOLOR :HIGHFREQ
IF :FREQ0 > 7040 [STOP]
TOOT 0 :FREQ0 15 10 TOOT 1 :FREQ1 15 7
SETPC 0 :PENCOLOR
SETPC 1 :PENCOLOR + 4
SETBG :SCREENCOLOR
FANFARE1 :FREQ0 * 2 :FREQ1 * 2
:PENCOLOR - 1 :SCREENCOLOR + 1
:HIGHFREQ

END

```

When FANFARE1 has been called for the final time, the game board is returned to its original colors.

Finally VICTORY asks DO YOU WANT TO PLAY AGAIN?. Here are the procedures, YES and NO, that are called by your response.

```
TO YES
PLAY
END
```

NO reinitializes Logo and clears the workspace.

```
TO NO
SETBG 74
SETPC 0 22
TELL [0 1 2 3] SETSH 0 CT
TELL 0 HOME ST SETPN 0
ERALL
RECYCLE
END
```

PROGRAM LISTING

```
TO PACGAME
SETUP
PLAY
END
```

```
TO SETUP
CS SS CT
PUT.SHAPE 1
SETUP.VARS
DRAW.BOARD
END
```

```
TO PUT.SHAPE :NUM
IF :NUM > 15 [STOP]
PUTSH :NUM THING WORD "SHAPE :NUM
ERN WORD "SHAPE :NUM
PUT.SHAPE :NUM + 1
END
```

```
TO SETUP.VARS
MAKE "270 7
MAKE "180 5
MAKE "90 3
MAKE "0 1
MAKE "POS3 [-95 -40]
MAKE "POS2 [-130 55]
MAKE "POS1 [116 100]
MAKE "PURPLE 70
MAKE "PACMAN 0
MAKE "BLACK 0
MAKE "TARGET 9
END
```

```
TO DRAW.BOARD
SETPC 0 22
SETPC 1 26
TELL 0 SETPN 0 PU HT
SETPOS [-160 120] PD SETH 90
REPEAT 10 [FD 320 SETY YCOR - 1]
REPEAT 40 [FD 230 SETPN 1 FD 70 SETPN ►
0 FD 20 SETY YCOR - 1]
REPEAT 60 [FD 20 SETPN 1 FD 280 SETPN ►
0 FD 20 SETY YCOR - 1]
FD 20 SETPN 1 FD 280 SETPN 0 FD 20
REPEAT 34 [FD 20 SETPN 1 FD 70 SETPN 0 ►
FD 58 SETPN 1 FD 30 SETPN 0 FD 142 ►
SETY YCOR - 1]
REPEAT 30 [FD 20 SETPN 1 FD 70 SETPN 0 ►
FD 230 SETY YCOR - 1]
REPEAT 15 [FD 320 SETY YCOR - 1]
PU
SETPOS [-105 -20] SETH 0 PD
FD 60 RT 90 FD 210 LT 90 FD 35
PU
END
```

```
TO PLAY
SET.PIECES
SET.DEMONS
END
```

```
TO SET.PIECES
MAKE "SCORE 0
```



```

TELL 0 HT HOME SETC 44
ASK [1 2 3] [HT PU SETC 7 SETSH ►
:TARGET]
ASK [1 2 3] [EACH [SETPOS THING WORD ►
"POS WHO] ST]
CT ST
MAKE "PAC.POSITION POS
END

TO SET.DEMONS
WHEN 7 [IF MEMBERP HEADING [0 90 180 ►
270] [ANIMATE.PACMAN] ►
[REVEAL.TURTLE]]
WHEN OVER 0 0 [STAY.IN.BOUNDS]
WHEN TOUCHING 0 1 [BULLSEYE 1]
WHEN TOUCHING 0 2 [BULLSEYE 2]
WHEN TOUCHING 0 3 [BULLSEYE 3]
END

TO ANIMATE.PACMAN
SETSH THING HEADING
WAIT 10
UPDATE
SETSH 1 + THING HEADING
END

TO UPDATE
IF COND OVER 0 1 [MAKE "PAC.POSITION ►
POS]
END

TO REVEAL.TURTLE
SETSH 0
SETCURSOR [0 19]
TYPE [ \ FIX HEADING TO GET BACK IN THE ►
GAME\ ]
END

TO STAY.IN.BOUNDS
TOOT 0 100 7 15
TOOT 1 200 7 15
SETPOS :PAC.POSITION
PR [] PR "OUCH!!! PR []
END

TO BULLSEYE :WHICH.TARGET
WHEN 7 [] SETC :BLACK
ADD.TO.SCORE
EXPLODE :WHICH.TARGET
( PR :SCORE [TO BE EXACT] )
WAIT 90
IF NOT :SCORE = 30000 ►
[ON.WITH.THE.GAME] [VICTORY]
END

```

```

TO ADD.TO.SCORE
MAKE "SCORE 10000 + :SCORE
CT PR [BOINK!!! WOWIEE!!!]
PR [SCORE LOTS OF POINTS FOR YOU!]
PR []
END

TO EXPLODE :TARGET
ASK :TARGET [EXPLODE1 [9 10 11 12 13 ►
12 13 12 13 12] [22 31 55 79 55 ►
79 55 79 55 7] HT HOME]
END

TO EXPLODE1 :SHAPES :COLORS
IF EMPTY :SHAPES [STOP]
TOOT 0 20 10 7 TOOT 1 25 10 6
SETSH FIRST :SHAPES
SETC FIRST :COLORS
EXPLODE1 BF :SHAPES BF :COLORS
END

TO ON.WITH.THE.GAME
TELL :PACMAN SETC :PURPLE
WHEN 7 [IF MEMBERP HEADING [0 90 180 ►
270] [ANIMATE.PACMAN] ►
[REVEAL.TURTLE]]
END

TO VICTORY
RECYCLE
TELL 0
CT
PR [THE WINNER, AND STILL CHAMP!]
FANFARE
WAIT 180
PR [DO YOU WANT TO PLAY AGAIN?]
END

TO FANFARE
FANFARE1 55 110 39 40 7040
FANFARE1 35 70 31 48 7040
FANFARE1 30 60 23 56 7040
FANFARE1 25 50 15 64 7040
FANFARE1 30 50 78 71 7680
FANFARE1 120 200 63 56 7680
SETPC 0 22
SETPC 1 26
SETBG 64
SETENV 0 15 SETENV 1 15
TOOT 0 240 15 240 TOOT 1 400 15 240
END

TO FANFARE1 :FREQ0 :FREQ1 :PENCOLOR ►
:SCREENCOLOR :HIGHFREQ

```

```

IF :FREQ0 > 7040 [STOP]
TOOT 0 :FREQ0 15 10 TOOT 1 :FREQ1 15 7
SETPC 0 :PENCOLOR
SETPC 1 :PENCOLOR + 4
SETBG :SCREENCOLOR
FANFARE1 :FREQ0 * 2 :FREQ1 * 2 ►
          :PENCOLOR - 1 :SCREENCOLOR + 1 ►
          :HIGHFREQ
END

TO YES
PLAY
END

TO NO
SETBG 74
SETPC 0 22
TELL [0 1 2 3] SETSH 0 CT
TELL 0 HOME ST SETPN 0
ERALL
RECYCLE
END

MAKE "SHAPE15 [0 0 0 0 0 0 0 0 0 0 0 0 ►
0 0 0 0]
MAKE "SHAPE14 [0 0 0 0 0 0 0 0 0 0 0 0 ►
0 0 0 0]

```

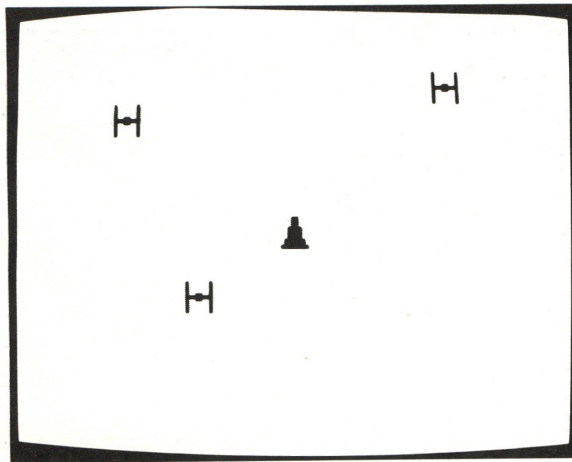
```

MAKE "SHAPE13 [129 66 36 153 90 36 90 ►
129 129 90 36 90 153 36 66 129]
MAKE "SHAPE12 [0 0 0 153 90 36 90 153 ►
153 90 36 90 153 0 0 0]
MAKE "SHAPE11 [0 0 0 0 24 60 102 219 ►
219 102 60 24 0 0 0 0]
MAKE "SHAPE10 [0 0 0 0 0 60 90 102 102 ►
90 60 0 0 0 0 0]
MAKE "SHAPE9 [255 129 129 129 129 129 ►
165 129 153 129 189 129 129 129 ►
129 255]
MAKE "SHAPE8 [0 0 0 0 28 62 62 127 7 ►
127 62 62 28 0 0 0]
MAKE "SHAPE7 [0 0 0 0 28 62 62 15 7 15 ►
62 62 28 0 0 0]
MAKE "SHAPE6 [0 0 0 0 0 0 56 124 254 ►
238 238 108 40 0 0 0]
MAKE "SHAPE5 [0 0 0 0 0 0 56 124 254 ►
238 198 198 130 0 0 0]
MAKE "SHAPE4 [0 0 0 0 56 124 124 254 ►
224 254 124 124 56 0 0 0]
MAKE "SHAPE3 [0 0 0 0 56 124 124 240 ►
224 240 124 124 56 0 0 0]
MAKE "SHAPE2 [0 0 0 0 40 108 238 238 ►
254 124 56 0 0 0 0]
MAKE "SHAPE1 [0 0 0 0 130 198 198 238 ►
254 124 56 0 0 0 0]

```

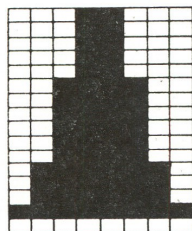
Blaster

That's your spaceship in the middle of the screen. You can steer with the joystick and fire lasers at the three enemy ships that surround you. You get points for hitting their ships. If an enemy ship collides with you, you lose a life. The game ends when you've lost five lives.

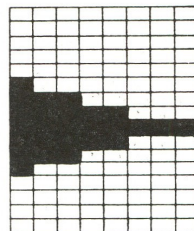


By Brian Harvey.

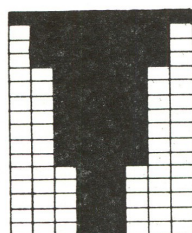
Turtle 0 represents your ship. It never moves from the center of the screen, but it can turn in different directions depending on the position of the joystick. It has eight possible shapes, each representing the ship facing one of the eight possible directions.



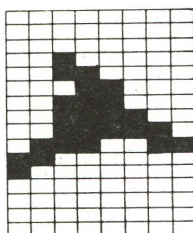
:SHIP1



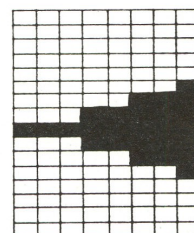
:SHIP3



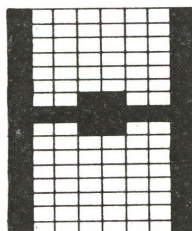
:SHIP5



:SHIP6

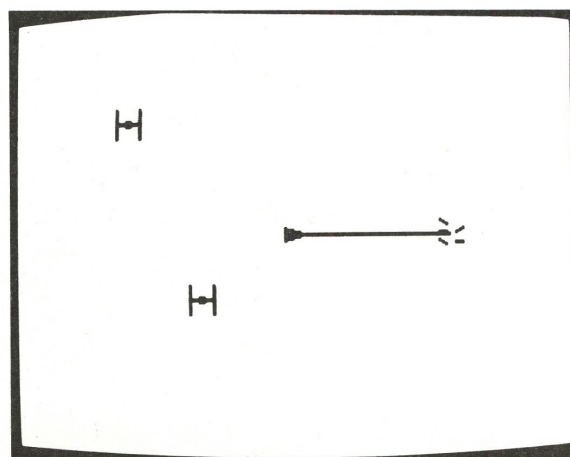
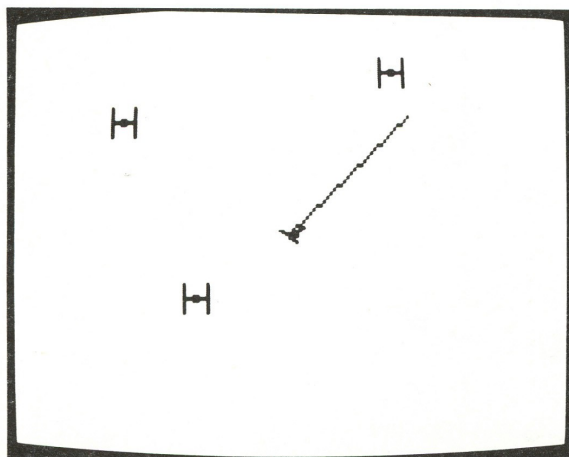


:SHIP7

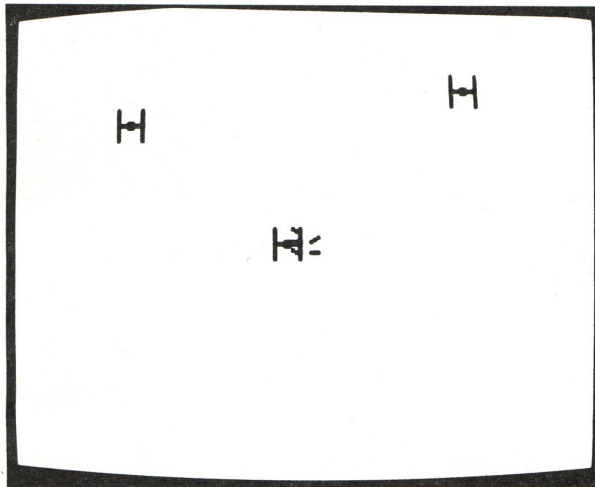


:BADGUY

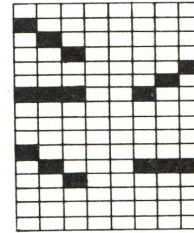
Turtles 1, 2, and 3 are the bad guys. They move at random speeds. Their direction is always more or less toward you, but not necessarily directly toward you. They have only one shape. You shoot by pressing the joystick button. This makes a red line appear for a moment, pointing in the direction the ship is facing. If this line hits one of the enemy ships, you score a point. Demons are used to detect the shot hitting an enemy.



Demons are also used to detect one of the enemy ships colliding with your ship. (The enemy ships don't fire at you; they have no weapons. All they can do is collide with you. Shame on you for firing at unarmed ships!) In the picture below, your ship has blown up because an enemy ship hit you.



LIVES:3 POINTS:14



:BLOWUP

To start the game, run the procedure BLASTER. It has two subprocedures, one to set up the screen and the other to play the game. The inputs to PLAY.BLASTER are the number of lives you're allowed and the number of points you start with.

```
TO BLASTER
  SETUP.BLASTER
  PLAY.BLASTER 5 0
END
```

Setting Up

The setup procedure sets colors and shapes, positions the turtles, and uses the PX command to set turtle 0 (your ship) in *penreverse* so that when you shoot, it can display and then erase the blast by retracing the line. Here is SETUP.BLASTER and its subprocedures.

```
TO SETUP.BLASTER
  CT
  SETBG 0
  SETPC 0 40
  PUTSHAPES 1 [SHIP1 SHIP2 SHIP3 SHIP4 SHIP5
               SHIP6 SHIP7 SHIP8 BADGUY BLOWUP]
  MAKE "ENEMIES [1 2 3]
  SETUP.ENEMIES
  SETUP.PLAYER
END
```

PUTSHAPES takes two inputs. The first input is the starting shape number. The second is a list of names containing shapes (in the list form output

GAMES

by GETSH). It uses PUTSH to copy those shapes into Logo's shape slots. In effect, this procedure replaces what would otherwise be ten individual PUTSH instructions.

```
TO PUTSHAPES :NUMBER :LIST
  IF EMPTY? :LIST [STOP]
  PUTSH :NUMBER THING FIRST :LIST
  PUTSHAPES :NUMBER+1 BF :LIST
END
```

SETUP.ENEMIES sets the shape and color of the enemy ships.

```
TO SETUP.ENEMIES
  TELL :ENEMIES
  HT
  SETSH 9
  SETC 74
  PU
END
```

SETUP.PLAYER sets the shape and color of your ship, and tells the turtle to penreverse, as explained earlier.

```
TO SETUP.PLAYER
  TELL 0
  ST
  SETSH 1
  SETC 7
  CS
  PX
END
```

Playing the Game

The main job of PLAY.BLASTER is to set up several demons. There is one for the joystick button, to fire a shot; three for the enemy ships colliding with pen 0, when you shoot them; three for the enemy ships colliding with turtle 0, when they hit you; and one for the joystick, to steer your ship. The procedure also puts the enemy ships in random positions, prints the initial score, and invokes BLASTER.LOOP to play the game.

Two variables are used throughout this part of the program to keep track of scoring. These variables are the two inputs to PLAY.BLASTER, called LIVES and POINTS.

LIVES The number of times an enemy ship can ram your ship before the game is over.

POINTS The number of times you've hit an enemy ship. This is your score.

Here are PLAY.BLASTER and its subprocedures.

```
TO PLAY.BLASTER :LIVES :POINTS
  SETUP.DEMONS
  TELL :ENEMIES
```

```

EACH [SETPOS RANDOM.POS]
SHOW.SCORE
ST
BLASTER.LOOP
END

```

SETUP.DEMONS starts the demons for the joystick button (firing), turtle-pen collisions (you shooting an enemy), turtle-turtle collisions (an enemy ramming you), and the joystick (steering).

```

TO SETUP.DEMONS
WHEN 3 [ASK 0 [FIRE]]
WHEN OVER 1 0 [ASK 1 [EXPLODE]]
WHEN OVER 2 0 [ASK 2 [EXPLODE]]
WHEN OVER 3 0 [ASK 3 [EXPLODE]]
WHEN TOUCHING 0 1 [ASK 0 [DIE 1]]
WHEN TOUCHING 0 2 [ASK 0 [DIE 2]]
WHEN TOUCHING 0 3 [ASK 0 [DIE 3]]
WHEN 15 [ASK 0 [STEER JOY 0]]
END

```

When nothing special is happening, the program spends most of its time in BLASTER.LOOP. It checks to see if you've run out of lives, in which case the game ends. Otherwise, it steers the enemy ships, shows the score, and continues. The ships are steered within 30 degrees of the direction toward you, so they tend to get closer to you but don't always move straight to you. (Their heading is chosen using the TOWARDS procedure, which is in the Towards and Arctan project.)

```

TO BLASTER.LOOP
IF :LIVES<1 [CS TS STOP]
EACH [SETH (TOWARDS [0 0])+(RANDOM 60)-30
      SETSP 30+RANDOM 150 WAIT RANDOM 30]
SHOW.SCORE
BLASTER.LOOP
END

```

When you move the joystick, a demon invokes the STEER procedure with turtle 0 active. The demon wakes up whenever the joystick is moved, including when it is returned to the center position. In that case, the input to STEER is -1 and nothing is done. Otherwise, we have to change the turtle's heading (so it can fire properly) and its shape.

```

TO STEER :WHERE
IF :WHERE<0 [STOP]
SETH 45*:WHERE
SETSH 1+:WHERE
END

```

When you push the joystick button, a demon invokes the FIRE procedure with turtle 0 active. This procedure draws and then erases a line representing your shot. It hides the turtle while drawing so that the ship doesn't appear to move.


```

TO FIRE
HT
FD 80
BK 80
ST
END

```

When your shot hits an enemy ship, a collision demon invokes the `EXPLODE` procedure, using `ASK` to make the turtle representing that ship become the current turtle. This procedure changes this turtle to an explosion shape, blinks it on and off, makes a noise, and then repositions the enemy ship somewhere else at random on the screen. It also adds one to your score.

```

TO EXPLODE
SETSH 10
TOOT 0 14 15 60
TOOT 1 16 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
SETPOS RANDOM.POS
SETSH 9
ST
MAKE "POINTS :POINTS+1
END

```

When an enemy ship hits your ship, a collision demon invokes the `DIE` procedure with turtle 0 active. The turtle number of the ship that hit you is an input to the procedure. That ship is moved to a random position, your ship explodes, the game stops for a second, and the count of how many lives remain is reduced by one. When your ship reappears, its shape is chosen to match its heading.

```

TO DIE :KILLER
ASK :KILLER [SETPOS RANDOM.POS]
SETSH 10
TOOT 0 25 15 60
TOOT 1 27 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
WAIT 60
SETSH 1+INT (HEADING/45)
ST
MAKE "LIVES :LIVES-1
END

```

Every so often, the `BLASTER.LOOP` procedure calls `SHOW.SCORE` to update the display of how many lives remain and how many points you've earned. This isn't done instantly when you get a point or lose a life, because it would slow down the play of the game. Because of this, you might sometimes get an extra (bonus) life if `BLASTER.LOOP` doesn't notice your death soon enough. That's why the procedure checks for a negative number of lives remaining and displays it as zero.

```

TO SHOW.SCORE
IF :LIVES<0 [MAKE "LIVES 0]
SETCURSOR [3 22]
TYPE SE "LIVES: :LIVES
SETCURSOR [20 22]
PRINT SE "POINTS: :POINTS
END

```

RANDOM.POS is the utility procedure that outputs a random position on the screen for use with SETPOS.

```

TO RANDOM.POS
OUTPUT LIST RANDOM 320 RANDOM 240
END

```

Shapes

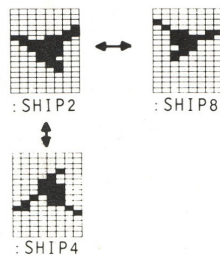
Here are the shapes.

```

MAKE "SHIP1 [24 24 24 24 24 60 60 60
        60 60 60 126 126 126 255 255]
MAKE "SHIP2 [0 0 0 0 1 3 206 124 60 28 24 12 4 0 0 0]
MAKE "SHIP3 [0 0 0 0 128 224 248 255
        255 248 224 128 0 0 0 0]
MAKE "SHIP4 [0 0 0 4 12 24 28 60 124 206 3 1 0 0 0 0]
MAKE "SHIP5 [255 255 126 126 126 60
        60 60 60 60 60 24 24 24 24 24]
MAKE "SHIP6 [0 0 0 32 48 24 56 60 62 115 192 128 0 0 0 0]
MAKE "SHIP7 [0 0 0 0 1 7 31 255 255 31 7 1 0 0 0 0]
MAKE "SHIP8 [0 0 0 0 128 192 115 62 60 56 24 48 32 0 0 0]
MAKE "BADGUY [129 129 129 129 129 129 153 255
        255 153 129 129 129 129 129 129]
MAKE "BLOWUP [128 192 96 32 1 3 230 228
        0 0 128 199 103 32 0 0]

```

Notice that SHIP4 to SHIP8 are simply mirror images of the first three ship shapes. For example, SHIP4 is the same as SHIP2 but reflected around a horizontal axis. SHIP8 is the same shape, but reflected around a vertical axis. These relationships can be seen in the lists of numbers representing the shape. For example, the list representing SHIP4 is the same as the list for SHIP2, but with the numbers in reverse order. (The relationship between a shape and its vertical-axis reflection is more complicated.) I actually only made the first three ship shapes in the shape editor; I created the others by calculating the appropriate numbers.



SUGGESTIONS

- Make the enemy ships fire back instead of just ramming you.
- The game hasn't been "playtuned." Should it be easier or harder? For example, the enemy ships move within 30 degrees of your direction. If that number were smaller, they'd hit you more often. The range of speeds could be changed too.
- You could start with a limited number of shots available. On the other hand, there could be a limit to the number of enemy ships that appear. (As it is, there is no way to "win" the game by destroying all the enemies.)
- You could add nice touches like stars in the background (remember to use a different pen for the stars and for the shots!) and sound effects between hits.
- It would be good to be able to move your own ship as well as steer it. This would require some way to indicate "thrust"; you could add a second joystick, or use the keyboard.

PROGRAM LISTING

The procedures from the Towards and Arctan project (p. 212) are also used in this program.

```

TO BLASTER
  SETUP.BLASTER
  PLAY.BLASTER 5 0
END

TO SETUP.BLASTER
  CT
  SETBG 0
  SETPC 0 40
  PUTSHAPES 1 [SHIP1 SHIP2 SHIP3 SHIP4 ►
    SHIP5 SHIP6 SHIP7 SHIP8 BADGUY ►
    BLOWUP]
  MAKE "ENEMIES [1 2 3]
  SETUP.ENEMIES
  SETUP.PLAYER
END

TO PUTSHAPES :NUMBER :LIST
  IF EMPTY? :LIST [STOP]
  PUTSH :NUMBER THING FIRST :LIST
  PUTSHAPES :NUMBER+1 BF :LIST
END

TO SETUP.ENEMIES
  TELL :ENEMIES
  HT
  SETSH 9
  SETC 74
  PU
END

TO SETUP.PLAYER
  TELL 0
  ST
  SETSH 1
  SETC 7
  CS
  PX
  END

TO PLAY.BLASTER :LIVES :POINTS
  SETUP.DEMONS
  TELL :ENEMIES
  EACH [SETPOS RANDOM.POS]
  SHOW.SCORE
  ST
  BLASTER.LOOP
END

TO SETUP.DEMONS
  WHEN 3 [ASK 0 [FIRE]]
  WHEN OVER 1 0 [ASK 1 [EXPLODE]]
  WHEN OVER 2 0 [ASK 2 [EXPLODE]]
  WHEN OVER 3 0 [ASK 3 [EXPLODE]]
  WHEN TOUCHING 0 1 [ASK 0 [DIE 1]]
  WHEN TOUCHING 0 2 [ASK 0 [DIE 2]]
  WHEN TOUCHING 0 3 [ASK 0 [DIE 3]]
  WHEN 15 [ASK 0 [STEER JOY 0]]
  END

```



```

TO BLASTER.LOOP
IF :LIVES<1 [CS TS STOP]
EACH [SETH (TOWARDS [0 0])+(RANDOM ►
    60)-30 SETSP 30+RANDOM 150 WAIT ►
    RANDOM 30]
SHOW.SCORE
BLASTER.LOOP
END

```

```

TO STEER :WHERE
IF :WHERE<0 [STOP]
SETH 45*:WHERE
SETSH 1+:WHERE
END

```

```

TO FIRE
HT
FD 80
BK 80
ST
END

```

```

TO EXPLODE
SETSH 10
TOOT 0 14 15 60
TOOT 1 16 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
SETPOS RANDOM.POS
SETSH 9
ST
MAKE "POINTS :POINTS+1
END

```

```

TO DIE :KILLER
ASK :KILLER [SETPOS RANDOM.POS]
SETSH 10
TOOT 0 25 15 60
TOOT 1 27 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
WAIT 60
SETSH 1+INT (HEADING/45)
ST
MAKE "LIVES :LIVES-1
END

```

```

TO SHOW.SCORE
IF :LIVES<0 [MAKE "LIVES 0]
SETCURSOR [3 22]
TYPE SE "LIVES: :LIVES
SETCURSOR [20 22]
PRINT SE "POINTS: :POINTS
END

```

```

TO RANDOM.POS
OUTPUT LIST RANDOM 320 RANDOM 240
END

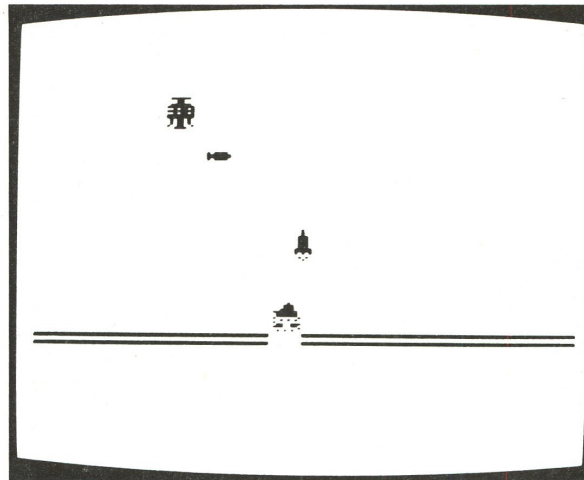
```

```

MAKE "SHIP1 [24 24 24 24 24 60 60 60 ►
    60 60 60 126 126 126 255 255]
MAKE "SHIP2 [0 0 0 0 1 3 206 124 60 28 ►
    24 12 4 0 0 0]
MAKE "SHIP3 [0 0 0 0 128 224 248 255 ►
    255 248 224 128 0 0 0 0]
MAKE "SHIP4 [0 0 0 4 12 24 28 60 124 ►
    206 3 1 0 0 0 0]
MAKE "SHIP5 [255 255 126 126 126 60 60 ►
    60 60 60 60 24 24 24 24]
MAKE "SHIP6 [0 0 0 32 48 24 56 60 62 ►
    115 192 128 0 0 0 0]
MAKE "SHIP7 [0 0 0 0 1 7 31 255 255 31 ►
    7 1 0 0 0 0]
MAKE "SHIP8 [0 0 0 0 128 192 115 62 60 ►
    56 24 48 32 0 0 0]
MAKE "BADGUY [129 129 129 129 129 129 ►
    153 255 255 153 129 129 129 129 ►
    129 129]
MAKE "BLOWUP [128 192 96 32 1 3 230 ►
    228 0 0 128 199 103 32 0 0]

```

Alien



Here is a description of this program by its author, Jeanry Chandler.

Alien is basically a Space Invaders-type game. All you need to play is a deft hand, and possibly a severe case of xenophobia. You control a sturdy defender tank with a joystick; you launch your deadly cruise missiles by (you guessed it) pressing the joystick button.

The alien craft, intent upon landing, will slip ever downward while avoiding your missiles and dropping its own neutro-destroyer bombs. If the alien lands, you are in serious trouble indeed. Two little green creatures will emerge and try to plant a bomb on your tank. You can attempt to shoot the little pests, but your gun has jammed and you can only shoot in one direction, so you have to shoot one quickly and then use the magic of Logoland to wrap and face the other. This, of course, is nearly impossible.

In this write-up I talk about the overall structure of the program and the decisions made about how it keeps track of things; I do not cover all the procedures in detail.

This game is in two parts. If you manage to shoot the alien helicopter before it lands, you don't play the second part. I have organized the program so that the two parts have the same structure.

Before proceeding further, you may want to play Alien. To begin the game, run `START`. You are the defender, controlling your maneuverable tank with the joystick plugged into port 1. You can switch the direction you are moving with the joystick and fire missiles at the alien helicopter with the joystick button.

Structure of the First Part of the Game

In their roles as alien, missile, bomb, and defender, the turtles are given their positions, headings, states, shapes, and colors. Demons are created to watch for certain game conditions and then call procedures that keep score, create explosions, and set variables. Those variables are:

SCORE	The score.
WIN	TRUE if something has happened that means the player has won the game.
LOSE	TRUE if something has happened that means the player has lost the game and thus has been annihilated.
MISSILE	Used to tell if a missile can be launched. Its value is 1 if the defender's missile is in the air, 0 if not. The rule is that the defender cannot launch a missile while the previous one is still in the air. When the missile gets to the top of the screen, it disappears and :MISSILE is reset to 0.
BOMB	Used to tell when to drop a bomb. After the alien drops a bomb, :BOMB becomes 1. This is used to prevent another from being dropped. When the bomb hits a target or the ground, :BOMB is reset to 0 to reenable launching.

It is a good idea to know the roles played by the four turtles.

- 0 Defender (the player)
- 1 Alien
- 2 Bomb (alien's neutro-destroyer bomb)
- 3 Missile (defender's missile)

Naming Conventions

Most procedures that deal with the alien have A in them; those that deal with the defender have D. The procedures that deal with the missile generally have MISSILE in their names, and the ones that deal with the alien's bomb have BOMB in theirs. The variables that have to do with the weapons are :MISSILE and :BOMB.

Setting Up

At the start of the game, SETUP initializes the turtle shapes and the game variables. Then it calls SETUP.DEMONS to create all the demons that are used in the game. SETUP then calls ASETUP and DSETUP to initialize the alien's and defender's positions, headings, colors, and speeds.

GAMES

```

TO SETUP
CT CS
SETUPSHAPES
TELL [0 1 2 3] HOME HT PU
TELL [1 0] ST PU
MAKE "BOMB 0
MAKE "MISSILE 0
MAKE "SCORE 0
MAKE "WIN "FALSE
MAKE "LOSE "FALSE
SETUP.DEMONS
DSETUP
ASETUP
TELL 0
END

```

```

TO ASETUP
TELL 1
SETSH 1 SETSP 70 SETH 90
SETPOS [0 100]
SETSH 3 SETSP 65 SETH 270
ASK 2 [SETC 44]
END

```

```

TO DSETUP
TELL 0 SETPOS [0 -55]
SETPN 0
RT 90 PD FD 300 PU RT 90 FD 5 RT 90 PD FD 300 PU
SETPOS [0 -43]
SETSP 75
END

```

Setting Up Demons and Demon Instruction Conventions

One of the demons that SETUP.DEMONS creates carries out its instructions every time the joystick position changes. (It is created by the line WHEN 15 [DMOVE]). The demon instructions call the procedure DMOVE to let the joystick control the defender's motion.

```

TO SETUP.DEMONS
WHEN 15 [DMOVE]
MISSILE.DEMONS
BOMB.DEMONS
END

```

```

TO DMOVE
IF MEMBERP JOY 0 [1 2 3] [ASK 0 [SETH 90]]
IF MEMBERP JOY 0 [5 6 7] [ASK 0 [SETH 270]]
END

```

The MISSILE.DEMONS and BOMB.DEMONS procedures create demons for actions having to do with the missile and bomb. For example:

```

WHEN TOUCHING 2 3 [SCORE 20 EXPLODE.BOMB.MISSILE]

```

is a line in `MISSILE.DEMONS` that creates a demon. This demon waits for a collision between the defender's missile and the alien's bomb. This demon's instructions make the missile explode, thus neutralizing the bomb and protecting the defender from it. `SCORE 20` gives the player points for having the good aim to hit the bomb. `EXPLODE.BOMB.MISSILE` makes the explosion graphics and sounds and sets `:MISSILE` and `:BOMB` to zero. This lets the game know that the missile and bomb have been destroyed.

`MISSILE.DEMONS` also creates a demon that waits for the joystick button to be pressed. When the button is pressed, a missile is fired.

```
TO MISSILE.DEMONS
WHEN 3 [IF :MISSILE < 1 [MISSILE]]
WHEN TOUCHING 2 3 [SCORE 20 EXPLODE.BOMB.MISSILE]
WHEN TOUCHING 3 1
    [EXPLODE.MISSILE.A SCORE 50 MAKE "WIN "TRUE]
WHEN OVER 3 0 [MAKE "MISSILE 0 ASK 3 [HT]]
END

TO EXPLODE.BOMB.MISSILE
ASK 3 [HT]
ASK 2 [SETSP 10]
ASK 2 [SETSH 6]
REPEAT 20 [TOOT 0 14 10 2]
ASK 2 [HT]
MAKE "BOMB 0
MAKE "MISSILE 0
END
```

The other explosion procedures are named for the things that cause each explosion. For example, the procedure that is invoked when the missile hits the alien is named `EXPLODE.MISSILE.A`. Remember that when this happens, the player wins the game. The same demon instructions that call `EXPLODE.MISSILE.A` also include `MAKE "WIN "TRUE`.

In this program, the demon instructions conventionally include a call to `SCORE` and a call to the appropriate explosion procedure. They also set `:WIN` or `:LOSE` if the game has been won or lost. For example, if the bomb hits the defender, the player has lost the game. The instructions run by the demon created for the collision between the bomb and the defender include the appropriate explosion procedure and also `MAKE "LOSE "TRUE`. This means you can systematically review all the conditions for winning and losing the game by reading through the demon setup procedures.

The Game Actions

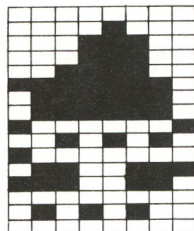
The main loop is `INVADE`. It stops only if the game has been won or lost. Either this happens in the first part of the game, or the alien helicopter survives to land and `INVADE` calls `LAND`. If `INVADE` calls `LAND`, then `INVADE` stops when the second part of the game is finished.

GAMES

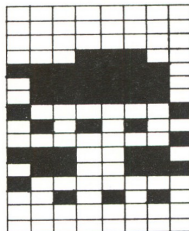
```

TO INVADE
IF :WIN [WIN STOP]
IF :LOSE [LOSE STOP]
TELL 0
SETSH :MISSILE + 8
BLADE
TELL 1
IF ( RANDOM 10 ) < 8 [SETY YCOR - 12]
IF YCOR < -45 [LAND STOP]
IF EQUALP RANDOM 3 1 [AMOVE]
IF EQUALP :BOMB 0 [BOMB]
BLADE
INVADE
END

```



Shape 8
:MISSILE = 0
:MISSILE + 8 = 8



Shape 9
:MISSILE = 1
:MISSILE + 8 = 9

INVADE calls most of the procedures that perform actions in the game. It does not call DMOVE or the explosion procedures; they are called by demons.

INVADE first checks :WIN and :LOSE to see if the end of the game has been signaled. If so, INVADE calls an appropriate procedure and stops. If not, INVADE updates the shape of the defender according to what is happening with the missile. It uses a trick with the shape numbers.

The trick is that INVADE uses :MISSILE to choose which shape to give the defender. :MISSILE gets changed by demon instructions. It is set to 1 when the missile is launched. This happens when the joystick button is pushed.* (See MISSILE.DEMONS.) When the missile hits something, or when it gets to the top of the screen, :MISSILE is set to 0 and the missile disappears. Demon instructions take care of this. While the missile is flying on the screen and :MISSILE is 1, INVADE gives the defender shape 9. When the missile is gone and :MISSILE is 0, INVADE gives the defender the ready-to-launch shape 8.

INVADE then calls BLADE, which makes the alien helicopter's rotor seem to turn by changing its shape. It also makes some sound effects to accompany the animation.

```

TO BLADE
TELL 1
SETSH 3
TOOT 0 80 12 1
WAIT 5
SETSH 10
TOOT 0 80 12 1
WAIT 5
SETSH 11
TOOT 0 80 12 1
WAIT 5
TELL 0
END

```

INVADE makes the alien drop closer to the ground 80 percent of the time. It uses (RANDOM 10) < 8 to decide whether to drop the alien.

*The MISSILE procedure fires the missile up and in the same general direction as the defender is traveling. It uses the ADJUST procedure to decide on the heading of the missile.

INVADE checks whether the alien has reached ground level. If it has, INVADE calls LAND, the second part of the game. Otherwise, INVADE may call AMOVE to make the alien change direction toward the defender. It uses $(\text{RANDOM } 3) = 1$ to do this one third of the time.

```
TO AMOVE
IF XCOR > ( ASK 0 [XCOR] ) [SETH 90] [SETH 270]
END
```

Next INVADE checks if the bomb is on the screen. If it is not, INVADE calls BOMB to launch one. BOMB aims the bomb at the defender using TOWARDS.*

```
TO BOMB
MAKE "BOMB 1
TELL 2
SETPOS ASK 1 [POS]
PU SETSH 5 POINT.AT.D SETSP 80 ST
END
```

Last, INVADE calls BLADE again to create more animation of the alien, and then calls itself to continue the game process.

Winning and Losing

The WIN and LOSE procedures print the score and either a congratulatory or a gloomy message.

The Second Part of the Game

If the alien craft survives your attacks and reaches ground level, LAND is called. LAND controls all the action that happens at ground level. Since this part of the game has a similar structure to the first part, the programs look similar. Some of the same shapes are used. There is still a defender (with the same shape), two aliens (with animated walking shapes), and a bullet for the defender to shoot.

The turtles' new assignments are

- 0 Defender (the player)
- 1 Green alien walker
- 2 Green alien walker
- 3 Bullet (the player's)

Since there are two alien walkers, and you have to shoot *both* of them to win the game, there is a new game variable AWCOUNT. :AWCOUNT is the number of aliens still alive. This lets the game know when you have shot an alien, and whether it is the last one.

LAND corresponds to START. It calls SETUP.LAND, which cancels all the demons created in the first part of the game and sets up the new shapes and turtle states. Then it calls SETUP.LAND.DEMONS, which creates the demons used in this part of the game. For example, the line WHEN 15 [DMOVE] in SETUP.LAND.DEMONS creates a demon that lets the joystick control the defender's motion.

*TOWARDS is described in Brian Harvey's project, Towards and Arctan.

GAMES

```

TO LAND
  SETUP.LAND
  SETUP.LAND.DEMONS
  WALK
END

```

```

TO SETUP.LAND
  CANCEL.DEMONS
  TELL [1 2]
  SETC 20
  SETSH 13
  SETY -43
  SETSP 70
  ST
  TELL 1 SETH 270
  TELL 2 SETH 90
  MAKE "AWCOUNT 2
END

```

```

TO CANCEL.DEMONS
  ASK 2 [HT CS]
END

```

```

TO SETUP.LAND.DEMONS
  WHEN 15 [DMOVE]
  WHEN TOUCHING 1 0 [EXPLODE.AW.D MAKE "LOSE "TRUE]
  WHEN TOUCHING 2 0 [EXPLODE.AW.D MAKE "LOSE "TRUE]
  WHEN TOUCHING 1 3 [EXPLODE.BULLET.AW 1]
  WHEN TOUCHING 2 3 [EXPLODE.BULLET.AW 2]
END

```

There is a new set of explosion procedures for this part of the game. It is possible that some of the explosion procedures from the invaders part of the game could have been reused; they might have had appropriate actions for the turtles' new roles. I decided that it would be clearer to have a new set of procedures with names that go with the new roles: AW for green alien walker and BULLET for the bullet.

The WALK procedure corresponds to INVADE in the first part of the game. WALK controls most of the game actions. The sequence of SETSH 13, T00T, and SETSH 12 creates the walking animation for turtles 1 and 2, the alien walkers. This sequence corresponds to BLADE in the first part of the game, which creates the helicopter blade animation. FIRE fires the bullet if the joystick button is pressed. It corresponds to MISSILE in the first part of the game. (I don't know why I put it in WALK instead of creating a demon to do it with WHEN 3 [FIRE].) Winning and losing the game happen exactly as in INVADE. WALK is recursive to keep the game process going.

```

TO WALK
  IF :WIN [WIN STOP]
  IF :LOSE [LOSE STOP]
  IF JOYB 0 [FIRE]
  TELL [1 2]
  SETSH 13

```



```

TOOT 0 300 8 4
SETSH 12
WALK
END

```

Note that there is nothing corresponding to the `MISSILE` or `BOMB` variables in this part of the game. The `BULLET` keeps going once it is fired. One bullet has to hit both alien walkers for the game to be won.

SUGGESTIONS

Jeanry suggests that you change the shapes that the turtles carry to make a different game. You might make a flying saucer blinking its landing lights instead of a helicopter spinning its blades.

A very different kind of game could be created with this type of programming. One idea is to replace the defender shooting destructive missiles with a ground launching platform trying to send recharge fuel cylinders to a disabled spaceship. The fuel could enable the spaceship to turn on its brakes and land safely instead of crashing. You could even make the joystick control the refueled spaceship, so that the new challenge is to land the ship.

You can use some of Alien's techniques—sound effects, controlling turtles with the joystick, simple game play—in projects completely of your own imagination.

PROGRAM LISTING

THE FIRST PART OF THE GAME

```

TO START
  SETUP
  INVAD
  END

TO SETUP
  CT CS
  SETUPSHAPES
  TELL [0 1 2 3] HOME HT PU
  TELL [1 0] ST PU
  MAKE "BOMB 0
  MAKE "MISSILE 0
  MAKE "SCORE 0
  MAKE "WIN "FALSE
  MAKE "LOSE "FALSE
  SETUP.DEMONS
  DSETUP
  ASETUP
  TELL 0
  END

```

```

TO INVAD
  IF :WIN [WIN STOP]
  IF :LOSE [LOSE STOP]
  TELL 0
  SETSH :MISSILE + 8
  BLADE
  TELL 1
  IF ( RANDOM 10 ) < 8 [SETY YCOR - 12]
  IF YCOR < -45 [LAND STOP]
  IF EQUALP RANDOM 3 1 [AMOVE]
  IF EQUALP :BOMB 0 [BOMB]
  BLADE
  INVAD
  END

```


DEMONS

```

TO SETUP.DEMONS
WHEN 15 [DMOVE]
MISSILE.DEMONS
BOMB.DEMONS
END

TO BOMB.DEMONS
WHEN TOUCHING 2 0 [EXPLODE.BOMB.D MAKE ▶
"LOSE "TRUE]
WHEN OVER 2 0 [EXPLODE.BOMB]
END

TO MISSILE.DEMONS
WHEN 3 [IF :MISSILE < 1 [MISSILE]]
WHEN TOUCHING 3 1 [EXPLODE.MISSILE.A ▶
SCORE 50 MAKE "WIN "TRUE]
WHEN OVER 3 0 [MAKE "MISSILE 0 ASK 3 ▶
[HT]]
END

```

SETUP FOR THE ALIEN AND DEFENDER

```

TO ASETUP
TELL 1
SETSH 1 SETSP 70 SETH 90
SETPOS [0 100]
SETSH 3 SETSP 65 SETH 270
ASK 2 [SETC 44]
END

TO DSETUP
TELL 0 SETPOS [0 -55]
SETPN 0
RT 90 PD FD 300 PU RT 90 FD 5 RT 90 PD ▶
FD 300 PU
SETPOS [0 -43]
SETSP 75
END

```

GAME ACTIONS

```

TO BLADE
TELL 1
SETSH 3
TOOT 0 80 12 1
WAIT 5
SETSH 10
TOOT 0 80 12 1
WAIT 5
SETSH 11

```

```

TOOT 0 80 12 1
WAIT 5
TELL 0
END

```

```

TO DMOVE
IF MEMBERP JOY 0 [1 2 3] [ASK 0 [SETH ▶
90]]
IF MEMBERP JOY 0 [5 6 7] [ASK 0 [SETH ▶
270]]
END

```

```

TO AMOVE
IF XCOR > ( ASK 0 [XCOR] ) [SETH 90] ▶
[SETH 270]
END

```

```

TO BOMB
MAKE "BOMB 1
TELL 2
SETPOS ASK 1 [POS]
PU SETSH 5 POINT.AT.D SETSP 80 ST
END

```

```

TO POINT.AT.D
SETH TOWARDS ASK 0 [POS]
END

```

```

TO MISSILE
MAKE "MISSILE 1
ASK 0 [SETSH 2]
ASK 3 [SETSH 4 SETX ( ASK 0 [XCOR] ) ▶
SEY -35 PU SETH ( ADJUST ( ASK 0 ▶
[HEADING] ) ) SETSP 95 ST]
END

```

```

TO ADJUST :HEADING
IF :HEADING = 90 [OP 75] [OP 285]
END

```

```

TO LOSE
PR [YOU HAVE BEEN ANNIHILATED !!!]
TELL [0 1 2 3] CS
END

```

```

TO WIN
PR [YOU WIN!!!!]
TELL [0 1 2 3] CS
END

```

EXPLOSIONS, SCORING, AND WINNING/LOSING

```

TO EXPLODE.BOMB
ASK 2 [SETSH 6]
REPEAT 2 [TOOT 0 20 15 5]
MAKE "BOMB 0
END

TO EXPLODE.BOMB.MISSILE
ASK 3 [HT]
ASK 2 [SETSP 10]
ASK 2 [SETSH 6]
REPEAT 20 [TOOT 0 14 10 2]
ASK 2 [HT]
MAKE "BOMB 0
MAKE "MISSILE 0
END

TO EXPLODE.MISSILE.A
TELL 3 HT
TELL 1 SETSH 7 SETSP 5
REPEAT 20 [TOOT 0 420 15 10]
TELL 1 HT
END

TO EXPLODE.BOMB.D
TELL 2 HT
TELL 0 SETSH 7 SETSP 5
REPEAT 40 [TOOT 0 30 15 5]
TELL 0 HT
END

```

SCORING PROCEDURES USED IN BOTH PARTS OF THE GAME

```

TO SCORE :S
MAKE "SCORE :SCORE + :S
PRINT SCORE
END

TO PRINT SCORE
PR SE [GOOD SHOT YOUR SCORE IS] :SCORE
END

```

THE SECOND PART OF THE GAME

```

TO LAND
SETUP.LAND
SETUP.LAND.DEMONS
WALK
END

```

```

TO SETUP.LAND
CANCEL.DEMONS
TELL [1 2]
SETC 20
SETSH 13
SETY -43
SETSP 70
ST
TELL 1 SETH 270
TELL 2 SETH 90
MAKE "AWCOUNT 2
END

```

DEMONS

```

TO CANCEL.DEMONS
ASK 2 [HT CS]
END

TO SETUP.LAND.DEMONS
WHEN 15 [DMOVE]
WHEN TOUCHING 1 0 [EXPLODE.AW.D MAKE ►
"LOSE "TRUE]
WHEN TOUCHING 2 0 [EXPLODE.AW.D MAKE ►
"LOSE "TRUE]
WHEN TOUCHING 1 3 [EXPLODE.BULLET.AW ►
1]
WHEN TOUCHING 2 3 [EXPLODE.BULLET.AW2]
END

```

GAME ACTIONS

```

TO WALK
IF :WIN [WIN STOP]
IF :LOSE [LOSE STOP]
IF JOYB 0 [FIRE]
TELL [1 2]
SETSH 13
TOOT 0 300 8 4
SETSH 12
WALK
END

TO FIRE
TELL 3
SETSH 14
SETH 270
ST SETSP 150 SETX ASK 0 [XCOR] SETY ►
-44
END

```

EXPLOSIONS, SCORING, AND WINNING/LOSING

```

TO EXPLODE.AW.D
TELL 0 GETSH 7 SETSP 5
REPEAT 40 [TOOT 0 30 15 5]
HT
END

TO EXPLODE.BULLET.AW :ALIEN
MAKE "AWCOUNT :AWCOUNT - 1
EXPLODE.B.AW :ALIEN
SCORE 30
IF :AWCOUNT = 0 [MAKE "WIN "TRUE]
END

```

```

TO EXPLODE.B.AW :A
ASK 3 [HT]
ASK :A [GETSH 7 SETSP 5]
REPEAT 10 [TOOT 0 420 15 10]
ASK :A [HT]
END

```

SETUPSHAPES, SAVE .SHAPES, AND SHAPES

```

TO SETUPSHAPES
PUTSH 1 :DEFENDER
PUTSH 2 :DEFENDER1
PUTSH 3 :ALIEN
PUTSH 4 :MISSILESHAPE
PUTSH 5 :BOMBSHAPE
PUTSH 6 :EXPLOSION1
PUTSH 7 :EXPLOSION2
PUTSH 8 :DEFENDER2
PUTSH 9 :DEFENDER3
PUTSH 10 :ALIEN2
PUTSH 11 :ALIEN3
PUTSH 12 :MAN1
PUTSH 13 :MAN2
PUTSH 14 :BULLET
END

```

```

TO SAVE.SHAPES
MAKE "DEFENDER GETSH 1
MAKE "DEFENDER1 GETSH 2

```

```

MAKE "ALIEN GETSH 3
MAKE "MISSILESHAPE GETSH 4
MAKE "BOMBSHAPE GETSH 5
MAKE "EXPLOSION1 GETSH 6
MAKE "EXPLOSION2 GETSH 7
MAKE "DEFENDER2 GETSH 8
MAKE "DEFENDER3 GETSH 9
MAKE "ALIEN2 GETSH 10
MAKE "ALIEN3 GETSH 11
MAKE "MAN1 GETSH 12
MAKE "MAN2 GETSH 13
MAKE "BULLET GETSH 14
END

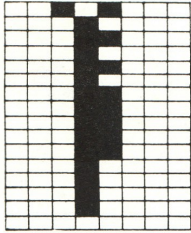
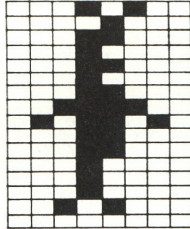
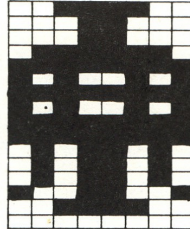
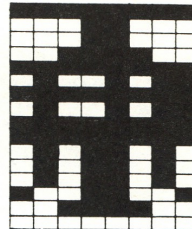
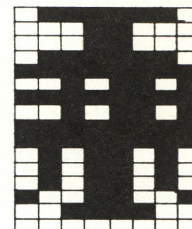
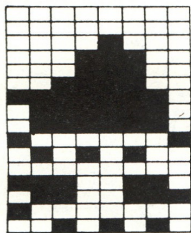
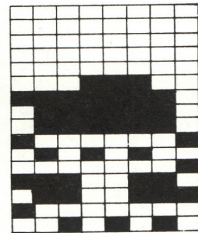
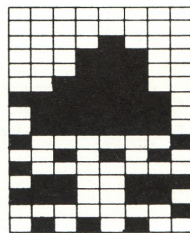
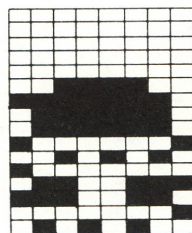
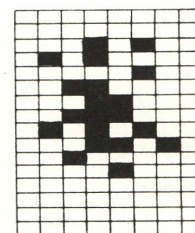
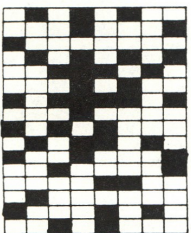
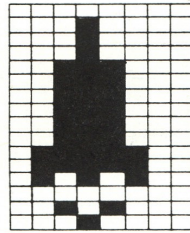
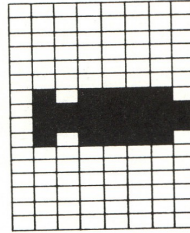
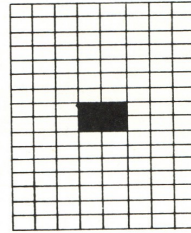
```

```

MAKE "BULLET [0 0 0 0 0 0 0 24 24 0 0 ▶
0 0 0 0]
MAKE "MAN2 [20 8 24 16 24 16 24 60 90 ▶
24 24 16 16 16 40 0]
MAKE "MAN1 [40 16 24 16 24 16 24 24 24 ▶
24 24 16 16 16 16 0]
MAKE "ALIEN3 [126 24 24 126 255 45 255 ▶
45 255 255 90 90 90 153 60 0]
MAKE "ALIEN2 [255 24 24 60 255 75 255 ▶
75 255 255 90 90 90 153 60 0]
MAKE "DEFENDER3 [0 0 0 0 0 60 254 126 ▶
126 129 42 128 103 230 1 84]
MAKE "DEFENDER2 [0 0 8 28 28 60 254 ▶
126 126 129 42 128 103 230 1 84]
MAKE "EXPLOSION2 [84 17 128 85 58 20 ▶
189 16 168 90 145 33 12 64 138 ▶
40]
MAKE "EXPLOSION1 [0 0 20 80 4 48 26 56 ▶
84 26 32 8 0 0 0 0]
MAKE "BOMBSHAPE [0 0 0 0 0 0 94 127 ▶
127 94 0 0 0 0 0 0]
MAKE "MISSILESHAPE [0 16 16 16 56 56 ▶
56 56 56 56 124 124 84 0 40 16]
MAKE "ALIEN [60 24 24 126 255 165 255 ▶
165 255 255 90 90 90 153 60 0]
MAKE "DEFENDER1 [0 0 0 0 0 28 254 126 ▶
126 129 84 1 230 103 128 42]
MAKE "DEFENDER [0 0 8 28 28 60 254 126 ▶
126 129 84 1 230 103 128 42]

```


SHAPES

:MAN1
slot 12:MAN2
slot 13:ALIEN
slot 3:ALIEN2
slot 10:ALIEN3
slot 11:DEFENDER
slot 1:DEFENDER1
slot 2:DEFENDER2
slot 8:DEFENDER3
slot 9:EXPLOSION1
slot 6:EXPLOSION2
slot 7:MISSILESHAPE
slot 4:BOMBSHAPE
slot 5:BULLET
slot 14

Adventure

Adventure is one of a class of hundreds, if not thousands, of games inspired by Crowther and Woods's classic FORTRAN program. You play an adventure game by exploring a simulated world, and usually you win points for finding objects, solving riddles, or killing monsters. This version, however, awards no points.

There are three aspects to an adventure program:

- Language understanding. The program must recognize commands that you give in a simple language.
- Simulation. The program executes your commands.
- Language production. The program tells you the results.

As you read on you'll find out how my program does all three of these things.

This adventure is smaller than most because of Logo's space limitations. Other microcomputer adventures are usually written in assembly language and also use a disk to store more information. On the other hand, this version was easy to write and is easy to modify and extend.

Adventure Programs Understand a Simple Language

When you play Adventure you give the computer commands in the Adventure language, just as when you use Logo you use the Logo language. Adventure is a program written in Logo that understands the Adventure language.*

Sentences in this language are in one of three forms:

- verb** Just a verb. The verb is one of these: LOOK, INVENTORY, or ?.
- verb noun** A verb followed by a noun, for example, TAKE KEY. Verbs are TAKE, DROP, EXAMINE, UNLOCK, or DRINK. Nouns are objects you find while playing the game.
- direction** The implied verb is MOVE, and the direction must be one of NORTH, EAST, SOUTH, WEST, UP or DOWN.

Most of the verbs have the same meaning as their English counterpart. (You can find out more by using them.)

The syntax rules of the language are simple and strict. Each verb is in one of two classes: either it must always be followed by a noun (TAKE KEY), or it must never be used with a noun. The program won't understand what you mean if you supply an object to a verb that doesn't expect one (for example, LOOK NORTH) or if you omit a necessary noun.

*Logo, the program that interprets the Logo language, is itself a program, written in the machine language of the microcomputer. For more on this subject, see the preface and Logo Interpreter project in this book.

Using the Program

To use the program, invoke the top-level procedure `ADVENTURE`.

The program describes the area you're in, then prompts you (with a `<`) for a sentence.

```
YOU ARE IN A FOREST
YOU SEE ASH AND BIRCH TREES
YOU CAN GO NORTH
YOU CAN GO EAST
YOU CAN GO SOUTH
YOU CAN GO WEST
>
```

You type in a sentence telling the program what to do. If it understands you it does what you typed; otherwise it complains. If your action takes you to a new place, the program describes the new place. After telling you the consequences of your action, the program is ready for another command.

```
>NORTH
YOU ARE AT THE BASE OF A CLIFF
YOU SEE ASH AND BIRCH TREES
YOU CAN GO SOUTH
YOU CAN GO WEST
A CLOSED STOUT IRON DOOR LEADS NORTH
>
```

If you move north, you'll find yourself at the foot of a cliff, with a door leading in. But to open the door you'll have to explore the forest further. Be careful not to get lost; in Adventure it isn't always true that if you travel east to get from one place to another that traveling west will get you back where you were.

You can get a list of every word the program knows by typing `?`.

You should probably play the game before reading further, because it is easier to understand the program if you've used it and because the game is much more fun if you don't know what to expect.

An Adventure Program Simulates a World

The program uses Logo *words* to represent the places and objects in the simulated world. The word `CROWBAR` represents the crowbar you'll find in the guard room. The word `GUARD` represents that guard room. There is one word for every place or object in the simulated world.

Objects and rooms in the simulation have different *attributes* (for example, weight). Each word has a list of all attributes of the thing it represents. A list of attributes is called a *property list*.*

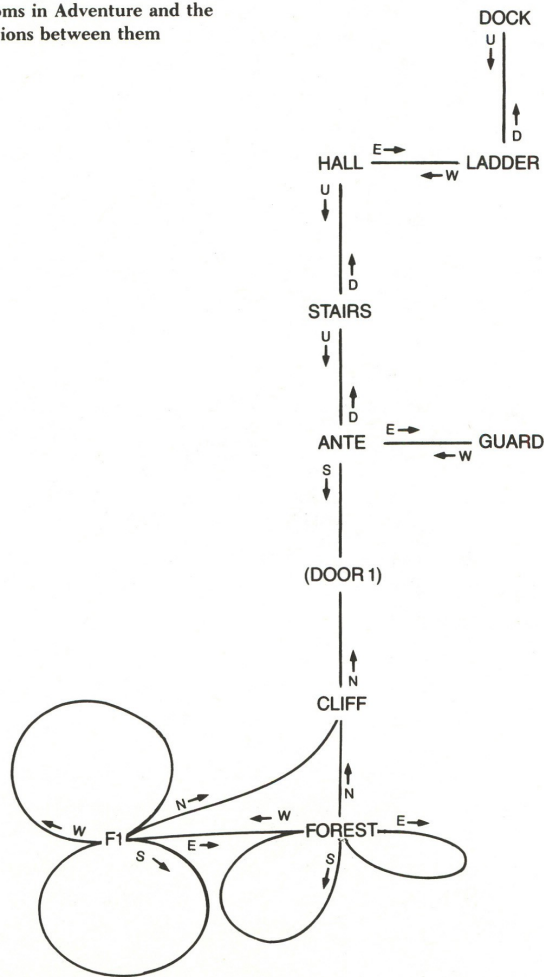
*This terminology comes from the programming language LISP. Some procedures in this project are tools for working with property lists: `PPROP`, `GPROP`, `HASPROP`, and `PROPTREE?`.

The Locales of the World Are Linked Rooms

Exploring in Adventure means moving from place to place and discovering objects in those places. The program thinks of all locales in the game (indoor rooms, the forest, stairways) as *rooms*. Each room is connected to at least one other room. You can move from one room to any connecting room—provided there's nothing preventing you from leaving, such as a shut door.

In Adventure there are six possible directions you can move: north, east, south, west, up, and down. Each room can have as many as six neighbors, one in each direction. This map shows how rooms are connected.

The rooms in Adventure and the connections between them



Each room is represented by a Logo word. Each word has an `EXITS` property that holds a list of six items, one for each possible exit direction from the room. Each item is the empty list if there is no exit in that direction; otherwise it is the word for the connecting room. Items in the

EXITS list appear in the order in which the direction options are presented: north, east, south, west, up, and down.

For example:

```
PPROP "STAIRS "EXITS [ [] [] [] [] ANTE HALL]
```

makes the room STAIRS have just two exits. (This makes sense, because we usually go up or down stairs, and there are exits at the top and bottom.) The up exit leads to ANTE, the down exit to HALL. (Actually, the exits lead to the rooms these words represent.)

I defined six variables to hold the positions of the exits in the exit list.

```
MAKE "NORTH 1
MAKE "EAST 2
MAKE "SOUTH 3
MAKE "WEST 4
MAKE "UP 5
MAKE "DOWN 6
```

Then I defined ITEM, an operation that outputs the *n*th element of a list. This made it possible for me to extract the exit from a room for any direction. For example, I can get the up exit from STAIRS with the following instruction:

```
PR ITEM :UP GPROP "STAIRS "EXITS
ANTE
```

```
PR FIRST BF BF BF BF BF GPROP "STAIRS "EXITS
```

The room exits are set up by INITROOMS. For a description, look at the listing at the end of this write-up.

Doors Were Hard to Add

In the outside world, nothing stops you from moving across open ground. But if you're in a building, there may be doors that stop you from getting into a room.* If a door is shut you must open it to get through, and if it is locked you must unlock it with a key before you can open it.

Doors are important in the real world, so I wanted to have doors in my program too. Doors were the most difficult part of the program to write. I tried a few different schemes before settling on one.

A door is a kind of exit from a room, but it isn't a destination in its own right. You may leave a room *through* a door, but you don't stay *in* the door. You go to the room on the other side.

Since doors are a type of exit, I decided that doors could go in the EXITS list just as rooms could. So I needed a predicate (DOORP) to distinguish doors from rooms, since both could be in the list. As you read on you'll discover other consequences of this decision.

*The word "indoors" is a reminder that one important thing about buildings is that they have doors, at least in most Western cultures.

You Leave a Room Through an Exit

When you move in a given direction, the program invokes the procedure MOVE. Its input is a number telling which way you want to move. (North is 1, east is 2, and so forth.)

```
TO MOVE :DIR
MOVE1 :DIR ( ITEM :DIR GPROP :#ROOM "EXITS )
END

TO MOVE1 :DIR :THERE
IF EMPTY? :THERE [PR [YOU CAN'T GO THAT WAY!] STOP]
IF DOORP :THERE [TRYDOOR :THERE :DIR] [GOROOM :THERE]
END
```

The global variable #ROOM holds the word representing the current room. I used a global variable because I knew I'd refer to it in many places in the program, and it would have been a bother to pass it as an input to all the procedures that need it.*

MOVE outputs an empty list if there is no exit; otherwise it outputs the word for the room or door in that direction.

If the exit is a door, the program uses TRYDOOR; if the exit is a room, the program uses GOROOM to put you in the new room.

```
TO GOROOM :NEW
MAKE "#ROOM :NEW
LOOK
END
```

GOROOM sets the global variable #ROOM to the new room and describes the locale.

Doors Are Tricky

The program knows which rooms are on both sides of any door because doors have EXITS properties just as rooms do. In a way, the program uses doors as if they were small rooms that you move through automatically. If leaving a room to the east takes you to an (open) door, you will go through the door to whatever is to the east of the door.

A word that stands for a door has a DOOR property on its property list. The value of its DOOR property is TRUE. The DOORP predicate checks this property:

```
TO DOORP :OBJ
OP PROPTURE? :OBJ "DOOR
END
```

A door also has a SHUT? property that is TRUE only if the door is shut;

*There are a few other global variables in the program. Almost all of them have names beginning with a sharp sign ("#") to distinguish them from procedure inputs. All global variables are set up by INITVARS at the start of the program.

a LOCKED? property that is TRUE only if the door is locked; and a KEY property that is the word representing the key that can unlock the door.

The procedure TRYDOOR tries to move you through a door in a certain direction. If the door is open, it finds the connected room by looking in the EXITS list for the door.

```
TO TRYDOOR :DOOR :DIR
IF GPROP :DOOR "SHUT? [PR [THE DOOR IS SHUT] STOP]
GOROOM ITEM :DIR GPROP :DOOR "EXITS
END
```

The adventure program in this project has only one door (DOOR1), but I designed it so that I could add more doors.

Adventure Programs Produce Language

The program prints descriptions of rooms and objects, tells you the results of things you do, and sometimes complains if you try something impossible. All the messages it prints are in fairly normal English.

The program describes what you'd see if you were really in the simulated world. When you enter a room or give the LOOK command, the program describes the room. When you give the INVENTORY command, the program describes whatever you're carrying. When you give the EXAMINE command, the program describes an object in more detail (sometimes).

Every object or room has a DESCRIPT property. For an object, this property is a noun phrase describing the object (for example, A DULL SWORD).* For a room, this property is a prepositional phrase describing your relation to the room (for example, IN A TWISTY MAZE OF LITTLE PASSAGES). The descriptions are in different forms because the descriptions are used in different ways. A room description is for telling you where you are (*in* a room, *on* a ladder, *at* a computer terminal) and the object descriptions are for saying what a thing is.

LOOK describes the room you're in.

```
TO LOOK
( PR [YOU ARE] GPROP :#ROOM "DESCRIPT )
IF NOT EMPTY GPROP :#ROOM "CONTAINS
[TYPE [YOU SEE] DESCRIBE GPROP :#ROOM "CONTAINS]
PREXITS
PRDOORS
END
```

The value of the CONTAINS property of a room is a list of the words for the objects in that room. If you look at INITROOMS (in the full listing), you'll see how I set up the initial contents of each room.

DESCRIBE prints the descriptions of a list of objects:

*The descriptions include the correct article (A or AN) for the word. I could have written a program to choose (checking whether the first letter is a vowel), but it would have taken up extra space and cost some extra time to execute. If there were seven hundred items instead of seven, I would have written the procedure, because it requires less space and less work to write it than to include an article in each of seven hundred descriptions.

GAMES

```

TO DESCRIBE :LIST
IF EMPTY :LIST [PR [] STOP]
TYPE "\
TYPE GPROP FIRST :LIST "DESCRIPT
IF NOT EMPTY BF :LIST [TYPE ",]
DESCRIBE BF :LIST
END

```

DESCRIBE gets each object's description from its DESCRIP property. It prints out the objects' descriptions, one after the other, all on the same line.

There are separate procedures for listing exits and doors because I thought it looked better to list them separately. PREXITS lists the directions in which you can leave a room, and PRDOORS lists the doors of the room. These procedures are similar.

```

TO PREXITS
PREXITS1 :#DIRNAMES GPROP :#ROOM "EXITS
END

```

```

TO PREXITS1 :DIRS :EXITS
IF EMPTY :DIRS [STOP]
PREXIT FIRST :DIRS FIRST :EXITS
PREXITS1 BF :DIRS BF :EXITS
END

```

The variable #DIRNAMES holds a list of the names of all directions in the same order as they appear in the exit list. PREXITS looks at the first direction name in #DIRNAMES and the first exit in the EXITS list. Since the names and the exits are in the same order, it can tell what name to use for the direction of the exit. It maintains this one-to-one correspondence as it checks each item of the lists. An item is an exit if it is not empty and not a door.

```

TO PREXIT :DIRECTION :OUT
IF EMPTY :OUT [STOP]
IF DOORP :OUT [STOP]
PR [YOU CAN GO] :DIRECTION
END

```

PRDOORS differs from PREXITS because it checks for doors instead of exits and tells you about the doors.

```

TO PRDOORS
PRDOORS1 :#DIRNAMES GPPROP :#ROOM "EXITS
END

```

```

TO PRDOORS1 :DIRS :DOORS
IF EMPTY :DIRS [STOP]
PRDOOR FIRST :DIRS FIRST :DOORS
PRDOORS1 BF :DIRS BF :DOORS
END

```



```

TO PRDOOR :DIRECTION :DOOR
IF EMPTY? :DOOR [STOP]
IF NOT DOORP :DOOR [STOP]
( PR IF GPROP :DOOR "SHUT? [[A CLOSED]] [[AN OPEN]]
GPROP :DOOR "DESCRIPT [LEADS] :DIRECTION)
END

```

The Syntax and Semantics of the Adventure Language

The first thing the program has to do to understand your sentence is to decide what type of word (noun, verb) each word in the sentence is. My program uses a very simple scheme: If the sentence is one word long, the first word must be a verb or a direction. If it is two words long, the first word must be a verb and the second a noun.

The procedure INTERPRET takes one input, a sentence.

```

TO INTERPRET :COM
IF ( COUNT :COM ) = 1 [INTER1 FIRST :COM STOP]
IF ( COUNT :COM ) = 2 [INTER2 FIRST :COM LAST :COM STOP]
PR [I KNOW ONLY ONE WORD AND TWO WORD SENTENCES.]
END

```

One-word and two-word sentences are handled by separate procedures. Anything else is an error.

```

TO INTER1 :VERB
IF MEMBERP :VERB :#DIRNAMES [MOVE THING :VERB STOP]
IF MEMBERP :VERB :#VERBS1 [RUN ( SE :VERB [] ) STOP]
PR [I DONT UNDERSTAND]
END

```

If you type NORTH, INTER1 finds the word NORTH in #DIRNAMES and knows you want to move in that direction. Each direction word has a value that is a number from 1 to 6 (an index into the EXITS list for the current room). The procedure uses THING to get the value of the direction word, and gives that as the input to the MOVE procedure.

Next the program looks at what the words *mean*. The "meaning" of a verb is given by a Logo procedure that carries out an action. For every verb in the language there is a Logo procedure. Conveniently, the Logo procedure has the same name as the verb.

The global variable #VERBS1 is a list of all single-word verbs.

```
MAKE "#VERBS1 [? LOOK INVENTORY]
```

If your single-word sentence is in #VERBS1, I use RUN to run the verb procedure. RUN wants a list as input, not a word, so I use SE to make a list containing only the verb.

INTER2 interprets two word sentences. Like INTER1, it checks whether the first word is a member of a list of verbs:

```
MAKE "#VERBS2 [TAKE DROP UNLOCK EXAMINE DRINK]
```

and runs a Logo procedure for the verb.

GAMES

```

TO INTER2 :VERB :NOUN
IF MEMBERP :VERB :#VERBS2
  [RUN LIST :VERB WORD "" :NOUN STOP]
PR [I DONT UNDERSTAND]
END

```

Each verb procedure takes one input, a noun. The noun is the second word in the sentence.

If you type the two-word sentence DRINK WINE, the program interprets the sentence by invoking the procedure DRINK (the verb) with the word WINE (the noun) as its input. In other words, the program carries out the Logo instruction

```
DRINK "WINE
```

That's why, in making the input list for RUN, I add the quote character (") in front of the noun. Otherwise Logo would try to run the instruction

```
DRINK WINE
```

which is wrong.

Verbs in One-word Sentences

The simplest verb is ?, which just prints the names of all verbs that the interpreter knows. It exists so you won't have to remember the names.

```

TO ?
PR :#VERBS1
PR :#VERBS2
PR :#DIRNAMES
END

```

The verb INVENTORY takes an inventory of objects you've picked up. The global variable #INVENTORY holds the list of objects. The procedure DESCRIBE (explained earlier) prints the actual description.

```

TO INVENTORY
TYPE [YOU'RE CARRYING]
IF EMPTY? :#INVENTORY [PR "\ NOTHING] [DESCRIBE :#INVENTORY]
END

```

The verb LOOK is called to describe a room when you enter it.

Verbs with Objects

Nouns refer to objects in the simulated world. The program has to determine what a noun means. A sentence like "take rope" means that the user wants to pick up the rope. Somehow the program has to translate the word "rope" to the word the program uses to represent the rope.

My program has an extremely simple solution. The word the program uses is the *same* as the word in the Adventure language. That is why I had to be careful choosing names for the words I used in the program. They had to be the same as the words I thought users would use in their sentences.

So if you say "take rope" the word "rope" can only mean the piece of rope that the word ROPE represents.*

Objects in Your Inventory or Locale

The predicate CARRYINGP tests whether you've got an object with you. You do if it's in your inventory, or if it is contained within an object in your inventory. (For example, if you're carrying a satchel and there is a blowtorch in the satchel, then CARRYINGP "BLOWTORCH" outputs TRUE.*

```
TO CARRYINGP :OBJ
OP WITHIN :#INVENTORY :OBJ
END
```

The predicate INROOMP tests whether an object is somewhere in the room.

```
TO INROOMP :OBJ
OP WITHIN GPROP :#ROOM "CONTAINS" :OBJ
END
```

WITHIN checks for an object either in a list or contained in an object in that list.

```
TO WITHIN :LIST :THING
IF EMPTY :LIST [OP "FALSE]
IF EQUALP :THING FIRST :LIST [OP "TRUE]
IF WITHIN :THING (GPROP FIRST :LIST "CONTAINS) [OP "TRUE]
OP WITHIN BF :LIST :THING
END
```

The EXAMINE Verb

Like all verbs with objects, EXAMINE first checks whether or not the object you mention is present by using ABSENT.*

```
TO ABSENT :OBJ
IF PRESENTP :OBJ [OP "FALSE]
PR [I DONT SEE IT HERE]
OP "TRUE
END
```

An object is present if it's either in your inventory or lying loose in the room.

*This solution has drawbacks. First, the user must spell the word exactly as I do and must not use synonyms. There are other drawbacks as well, but I'll save them for later.

I put this feature in, even though there are no carryable containers in the game, because it seemed elegant, and I might want to add containers later.

ABSENT combines two actions in one procedure. It is a predicate, the opposite of PRESENTP, and it also prints a message if the object is absent.

This extra action restricts the usefulness of ABSENT. It should only be called by a verb procedure, because otherwise it is not appropriate to print the message. Usually it's a bad idea to combine functions like this, but I did it after I discovered that only verb procedures used ABSENT and that each of them printed the same message if the object was absent. I combined the test and the message into one procedure to save space.

GAMES

```

TO PRESENTP :OBJ
OP OR CARRYINGP :OBJ INROOMP :OBJ
END

```

If an object is absent, the program just says so. The program deliberately doesn't distinguish between objects that exist somewhere but aren't nearby and objects that don't exist. If the object contains something, EXAMINE tells you about it. That's the only detail that EXAMINE ever gives.

```

TO EXAMINE :OBJ
IF ABSENT :OBJ [STOP]
IF EMPTY GPROP :OBJ "CONTAINS [STOP]
TYPE [IT HOLDS] DESCRIBE GPROP :OBJ "CONTAINS
END

```

The TAKE and DROP Verbs Change Your Inventory

The TAKE program has to be more careful than EXAMINE. This is because there are many reasons that might prevent you from taking an object.

- It might not be there.
- You might already have it.
- It might be too heavy.
- It might be impossible to carry.
- Your arms could be full.

The program checks for each of these, making an appropriate complaint. If nothing prevents it, the object is added to your inventory and removed from the contents of the room.

```

TO TAKE :OBJ
IF ABSENT :OBJ [STOP]
IF CARRYINGP :OBJ [PR [YOU ALREADY HAVE IT!] STOP]
IF PROPTIME? :OBJ "IMMOBILE? [PR [IT'S TOO HEAVY] STOP]
IF PROPTIME? :OBJ "LIQUID? [PR [NO CONTAINER] STOP]
IF (COUNT :#INVENTORY) > 2 [PR [YOUR BAG IS FULL!] STOP]
MAKE "#INVENTORY FPUT :OBJ :#INVENTORY
PPROP :#ROOM "CONTAINS (REMOVE :OBJ GPROP :#ROOM "CONTAINS)
END

```

When I wrote this procedure I had to decide how to represent the mobility of objects. I could have given everything a WEIGHT property and compared that with a STRENGTH variable, but I rejected that as too much work. All I wanted was to prevent clearly impossible requests, such as picking up trees. For my purposes, objects are either heavy or not, so this suggested a property that was TRUE only if the object was liftable.

I chose to give heavy objects an IMMOBILE? property of TRUE instead of giving light objects a MOBILE? property because I knew I could save some space that way. I knew there would be only a few heavy objects and many light ones, and if I wrote my programs to assume that an object was light unless explicitly marked heavy I could avoid storing all the IMMOBILE? properties that were FALSE.

To make this assumption easier to program, I wrote the predicate `PROPTTRUE?`, which outputs `TRUE` only if the value of the property is `TRUE`. If an object doesn't have any value for a certain property, then `GPROP` outputs the empty list. The empty list isn't `TRUE` or `FALSE`, so I can't use `GPROP` directly in an `IF`.

```
TO PROPTTRUE? :OBJ :PROP
OP ( GPROP :OBJ :PROP ) = "TRUE
END
```

I used the same kind of reasoning in defining the `LIQUID?` property. Most objects are dry.

It took a lot of thought to save a little space. Fortunately, the program is only a little harder to understand as a result. It would be foolish to make the program very complex just to save a little space.

`DROP` is the opposite of `TAKE`, but has to check only whether you are carrying the object.

```
TO DROP :OBJ
IF NOT CARRYINGP :OBJ [PR [YOU'RE NOT CARRYING IT!] STOP]
MAKE "#INVENTORY REMOVE :OBJ :#INVENTORY
PPROP :#ROOM "CONTAINS ( FPUT :OBJ GPROP :#ROOM "CONTAINS )
END
```

`REMOVE` takes as inputs a list and an item in the list and outputs the list with the item removed.

```
TO REMOVE :THING :LIST
IF EMPTY :LIST [OP []]
IF :THING = FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST REMOVE :THING BF :LIST
END
```

The Game Program

Before showing you the rest of the verbs, I'd like to show you the top-level game loop. You start the program with `ADVENTURE`.

```
TO ADVENTURE
INIT
LOOK
GAMELOOP
END
```

`INIT` initializes everything.

```
TO INIT
INITVARS
INITOBS
INITROOMS
END
```

`INITVARS` initializes all global variables, `INITOBS` initializes all objects, and `INITROOMS` initializes all rooms. I won't include their definitions

GAMES

here because they are just long lists of MAKEs and PPROP. They are in the listing at the end of this write-up.

LOOK describes the room you're in. It was explained earlier.

The game loop is GAMELOOP:

```
TO GAMELOOP
  INTERPRET GETINPUT
  IF NOT EMPTY :#RESULT [PR :#RESULT STOP]
  GAMELOOP
END
```

GETINPUT prompts for a sentence and returns it.

```
TO GETINPUT
  TYPE ">"
  OP RL
END
```

The game loop runs until the variable #RESULT becomes nonempty. Any verb can end the game by putting a message about the result of the game into that variable.

Now we'll look at the last two verbs.

UNLOCK *Unlocks a* DOOR

Although it may not appear so at first, it's a little difficult for the program to understand UNLOCK DOOR because it's hard to tell what object the word DOOR refers to. Remember, I wanted it to be possible for there to be many doors.

The UNLOCK verb first ensures that you asked to unlock a DOOR. The program then uses GETDOOR to try to find a door. If there is one, UNLOCK looks at its KEY property to be sure that you have the key that unlocks it:

```
TO UNLOCK :OBJ
  IF NOT EQUALP :OBJ "DOOR [PR [I CANT UNLOCK THAT] STOP]
  MAKE "OBJ GETDOOR ( GPROP :#ROOM "EXITS )
  IF EMPTY :OBJ [PR [NOTHING UNLOCKABLE HERE] STOP]
  IF NOT CARRYINGP GPROP :OBJ "KEY [PR [YOU CAN'T] STOP]
  PPROP :OBJ "LOCKED? "FALSE
  PPROP :OBJ "SHUT? "FALSE
END
```

Unlocking a door also opens it automatically; you don't need to OPEN a door after you UNLOCK it.*

The procedure GETDOOR looks at every item in the EXITS list of the current room until it finds one that is a door, and outputs it. This door is assumed to be the object referred to by "DOOR" in the sentence "UNLOCK DOOR".

```
TO GETDOOR :EXITS
  IF EMPTY :EXITS [OP []]
```

*I didn't have enough space for a verb OPEN, and requiring you to OPEN the door slows the game down to no purpose, anyway.

```
IF DOORP FIRST :EXITS [OP FIRST :EXITS]
OP GETDOOR BF :EXITS
END
```

DRINKing *Ends the Game*

If you've played the game, you know the unfortunate effects of drinking wine.

```
TO DRINK :OBJ
IF ABSENT :OBJ [STOP]
IF NOT PROPTURE? :OBJ "LIQUID? [PR [NOT A LIQUID!] STOP]
PR [YOU DRINK THE WINE...]
WAIT 10
PR [IT'S DELICIOUS...] WAIT 10
PR [AND VERY POTENT. YOU GET DRUNK, AND FALL IN THE RIVER.]
MAKE "#RESULT [YOU DROWNED]
END
```

The DRINK verb is a little strange. It checks that you are drinking a liquid that is present, but then it makes two assumptions: that you are drinking wine and that you are drinking by the river. The program tells you the deadly result by setting the global variable #RESULT to a sentence. GAMELOOP notices, and ends the game.

The two assumptions are used in at least two ways. The first is that the message uses the words "wine" and "river" explicitly, and also says that the result is drunkenness. The second use of the assumption is that the result only is possible if you are by the river.

The assumption that you drank wine must be true because the only liquid in the game is wine. The assumption of locale is safe because the only wine in the game is in the barrel, and you can't move the barrel.

It is not a good thing to make assumptions like these in writing programs because it makes it hard to extend the program. (If I had added other liquids, I would have had to add an INTOXICATING? property to WINE to distinguish it from safer liquids.) I did it to save space, but I'm not proud of it.

You Can Change This Adventure in Many Ways

The easiest thing to do is to add new rooms. All you need to do is change the property assignments in INITROOMS. Make sure that you provide some path from every room to every other room.

You can add new doors in the same way as you add rooms. But it's hard to add a new key, for reasons I'll explain. So when making new doors, either they should not be locked or they should use the same key.

If you want to make a one-way exit from one room to another, do not include the first room in the EXITS of the second. (You can also make one-way doors.)

It's also easy to put new objects into the game. All the objects are created by INITOBS.

You add new verbs by writing the procedure and modifying the list #VERBS1 or #VERBS2. But new verbs or objects may need some new properties. For example, if you added the verb EAT you'd want to give edible objects a FOOD? property of TRUE, and have EAT check it.

It might be fun to make a type of door that only opens after you take some action such as saying a password, or pressing a button.

Your verbs can end the game at any time by setting #RESULT.

It's easy to debug changes to Adventure. You can stop the program with the BREAK key, look at things, fix them, then resume with GAMELOOP. (This is also a good way to cheat.)

Some Problems with My Program

The scheme I use for semantics is not very good. The nouns you type must be the same words as those used by the program. That is why it would be hard to add another key (for example, a brass one). What Logo word would you use to represent it? You can't use KEY, because that word already stands for a different key, the iron one in the forest. When you type TAKE KEY the program looks for the word spelled "k-e-y". Suppose you use KEY1. The description of the new key would be [A BRASS KEY], and the user would try to refer to it with the word used in the description, namely KEY. The user would have no way to know that the "right" word is really KEY1, and even if the program printed out that word, it wouldn't be much like English. Can you imagine "You see a brass key1"?

For the same reason, there cannot be a second sword, or crowbar, or any such thing. The problem is most acute with keys, though, because that means that one key must be able to unlock all doors.

Note that this is not a problem for rooms, because the user never refers to a room in any way. It is also not much of a problem for verbs, because it would be easy to give each verb a property holding the name of a Logo procedure to run. Then verbs would not have to have the same name as the procedure that defines them.

One possible fix to this would be to give each item a property for what "kind" of thing it is.

```
PPROP "KEY "KIND "KEY
PPROP "KEY1 "KIND "KEY
```

Then a reference to a "key" could be interpreted as meaning *any* object that was a key. This is similar to the way doors are identified by GETDOOR.

Another possible solution would be to give the noun KEY a property list of all the program words that are a "kind of" key.

There would still be problems with ambiguity. There might be two keys in the area. There is no way at present for the program to ask the user to say which key was meant. (This is a problem with GETDOOR as well. It takes the first door.) Perhaps the program could ask:

```
>TAKE KEY
DO YOU MEAN THE BRASS KEY, OR THE IRON KEY? BRASS
OK
```

Another problem is that room descriptions sometimes refer to things that the user might mention. For example, the description of the dock mentions an underground stream. People often try to drink the water, but the program doesn't even know there is a "stream" nearby, much less that a stream holds "water." The word "stream" is contained inside the description, and the program has no way to use it other than by printing it.

If scenes were described by properties, descriptions could perhaps be built from them, and the program would have access to the properties of the room. But generating good English from a set of properties is a difficult problem.

Some Adventuresome Improvements

There are many possible improvements to this game, some easy, others more difficult.

Writing programs that understand and produce natural language is a challenge for hundreds of researchers throughout the world. In a small way, Adventure is a part of this research.

First, you could fix the problems I just mentioned. But there is even more to do.

Consider a dialogue like

```
YOU ARE IN A FOOD STORE  
YOU SEE A GREEN CHEESE  
>TAKE IT
```

The program could use context to figure out what the word "it" means.

Or suppose you're carrying a baseball bat and a rock and are attacked by a vampire bat. The word "bat" in HIT BAT means the vampire bat, not the baseball bat.

It would also be very nice to have a richer syntax than the simple verb and noun scheme used here.

Many adventure games have autonomous characters. Usually they are your foes. It would be a fine challenge to add them to this game. Characters should move from room to room on their own, and sometimes the player should encounter them. The results need not always be woeful.

More complexly structured worlds are possible. The objects in my world are mostly decorative—there is nothing to pry with the crowbar, nowhere to climb with the rope.

If the Adventure language was extended such that you could use it to program, then you could teach a turtle how to explore, send it in to dangerous areas, and have it carry back things for you.

Writing good adventure programs is an art and a game of its own. Now that you've explored the simulated world of Adventure, perhaps it's time for you to begin exploring Adventure itself.

PROGRAM LISTING

```

TO ADVENTURE
INIT
LOOK
GAMELOOP
END

TO INIT
INITVARS
INITOBS
INITROOMS
END

TO GAMELOOP
INTERPRET GETINPUT
IF NOT EMPTY :#RESULT [PR :#RESULT ►
    STOP]
GAMELOOP
END

TO GETINPUT
TYPE ">"
OP RL
END

TO INTERPRET :COM
IF ( COUNT :COM ) = 1 [INTER1 FIRST ►
    :COM STOP]
IF ( COUNT :COM ) = 2 [INTER2 FIRST ►
    :COM LAST :COM STOP]
PR [I KNOW ONLY ONE WORD AND TWO WORD ►
    SENTENCES.]
END

TO INTER1 :VERB
IF MEMBERP :VERB :#DIRNAMES [MOVE ►
    THING :VERB STOP]
IF MEMBERP :VERB :#VERBS1 [RUN SE ►
    :VERB [] STOP]
PR [I DONT UNDERSTAND]
END

TO INTER2 :VERB :NOUN
IF MEMBERP :VERB :#VERBS2 [RUN LIST ►
    :VERB WORD "" :NOUN STOP]
PR [I DONT UNDERSTAND]
END

TO MOVE :DIR
MOVE1 :DIR ( ITEM :DIR GPROP :#ROOM ►
    "EXITS )
END

TO MOVE1 :DIR :THERE
IF EMPTY :THERE [PR [YOU CAN'T GO ►
    THAT WAY!] STOP]
IF DOORP :THERE [TRYDOOR :THERE :DIR] ►
    [GOROOM :THERE]
END

TO DOORP :OBJ
IF EMPTY :OBJ [OP "FALSE]
OP PROPTRUE? :OBJ "DOOR?
END

TO GOROOM :NEW
MAKE "#ROOM :NEW
LOOK
END

TO LOOK
( PR [YOU ARE] GPROP :#ROOM "DESCRIP ►
    )
IF NOT EMPTY GPROP :#ROOM "CONTAINS ►
    [TYPE [YOU SEE] DESCRIBE GPROP ►
        :#ROOM "CONTAINS]
PREXITS
PRDOORS
END

TO DESCRIBE :LIST
IF EMPTY :LIST [PR [] STOP]
TYPE "\
TYPE GPROP FIRST :LIST "DESCRIP
IF NOT EMPTY BF :LIST [TYPE ",]
DESCRIBE BF :LIST
END

TO PREXITS
PREXITS1 :#DIRNAMES GPROP :#ROOM ►
    "EXITS
END

TO PREXITS1 :DIRS :EXITS
IF EMPTY :DIRS [STOP]
PREXIT FIRST :DIRS FIRST :EXITS
PREXITS1 BF :DIRS BF :EXITS
END

TO PREXIT :DIRECTION :OUT
IF EMPTY :OUT [STOP]
IF DOORP :OUT [STOP]
( PR [YOU CAN GO] :DIRECTION )
END

```



```

TO PRDOORS
PRDOORS1 :#DIRNAMES GPROP :#ROOM ►
    "EXITS
END

TO PRDOORS1 :DIRS :DOORS
IF EMPTY :DIRS [STOP]
PRDOOR FIRST :DIRS FIRST :DOORS
PRDOORS1 BF :DIRS BF :DOORS
END

TO PRDOOR :DIRECTION :DOOR
IF NOT DOORP :DOOR [STOP]
( PR IF PROPTRUE? :DOOR "SHUT? [[A ►
    CLOSED]] [[AN OPEN]] GPROP :DOOR ►
    "DESCRIPT [LEADS] :DIRECTION )
END

TO TRYDOOR :DOOR :DIR
IF GPROP :DOOR "SHUT? [PR [THE DOOR IS ►
    SHUT] STOP]
GOROOM ITEM :DIR GPROP :DOOR "EXITS
END

TO ?
PR :#VERBS1
PR :#VERBS2
PR :#DIRNAMES
END

TO INVENTORY
TYPE [YOU'RE CARRYING]
IF EMPTY :#INVENTORY [PR "\ NOTHING] ►
    [DESCRIBE :#INVENTORY]
END

TO TAKE :OBJ
IF ABSENT :OBJ [STOP]
IF CARRYINGP :OBJ [PR [YOU ALREADY ►
    HAVE IT!] STOP]
IF PROPTRUE? :OBJ "IMMOBILE? [PR [ITS ►
    TOO HEAVY] STOP]
IF PROPTRUE? :OBJ "LIQUID? [PR [NO ►
    CONTAINER] STOP]
IF ( COUNT :#INVENTORY ) > 2 [PR [YOUR ►
    BAG IS FULL!] STOP]
MAKE "#INVENTORY FPUT :OBJ :#INVENTORY
PPROP :#ROOM "CONTAINS ( REMOVE :OBJ ►
    GPROP :#ROOM "CONTAINS )
END

TO CARRYINGP :OBJ
OP WITHIN :#INVENTORY :OBJ
END

```

ADVENTURE

189

```

TO ABSENT :OBJ
IF PRESENTP :OBJ [OP "FALSE]
PR [I DONT SEE IT HERE]
OP "TRUE
END

TO PRESENTP :OBJ
OP OR CARRYINGP :OBJ INROOMP :OBJ
END

TO WITHIN :LIST :THING
IF EMPTY :LIST [OP "FALSE]
IF EQUALP :THING FIRST :LIST [OP ►
    "TRUE]
IF WITHIN ( GPROP FIRST :LIST ►
    "CONTAINS ) :THING [OP "TRUE]
OP WITHIN BF :LIST :THING
END

TO INROOMP :OBJ
OP WITHIN GPROP :#ROOM "CONTAINS :OBJ
END

TO DROP :OBJ
IF NOT CARRYINGP :OBJ [PR [YOUR NOT ►
    CARRYING IT!] STOP]
MAKE "#INVENTORY REMOVE :OBJ ►
    :#INVENTORY
PPROP :#ROOM "CONTAINS ( FPUT :OBJ ►
    GPROP :#ROOM "CONTAINS )
END

TO EXAMINE :OBJ
IF ABSENT :OBJ [STOP]
IF EMPTY GPROP :OBJ "CONTAINS [STOP]
TYPE [IT HOLDS] DESCRIBE GPROP :OBJ ►
    "CONTAINS
END

TO UNLOCK :OBJ
IF NOT EQUALP :OBJ "DOOR [PR [I CANT ►
    UNLOCK THAT] STOP]
MAKE "OBJ GETDOOR ( GPROP :#ROOM ►
    "EXITS )
IF EMPTY :OBJ [PR [NOTHING UNLOCKABLE ►
    HERE] STOP]
IF NOT CARRYINGP GPROP :OBJ "KEY [PR ►
    [YOU CAN'T] STOP]
PPROP :OBJ "LOCKED? "FALSE
PPROP :OBJ "SHUT? "FALSE
END

```

```

TO GETDOOR :EXITS
IF EMPTY :EXITS [OP []]
IF DOORP FIRST :EXITS [OP FIRST ►
:EXITS]
OP GETDOOR BF :EXITS
END

TO DRINK :OBJ
IF ABSENT :OBJ [STOP]
IF NOT PROPTURE? :OBJ "LIQUID? [PR ►
[NOT A LIQUID!] STOP]
PR [YOU DRINK THE WINE...]
WAIT 10
PR [IT'S DELICIOUS...] WAIT 10
PR [AND VERY POTENT. YOU GET DRUNK, ►
AND FALL IN THE RIVER.]
MAKE "#RESULT [YOU DROWNED]
END

TO INITVARS
MAKE "NORTH 1
MAKE "EAST 2
MAKE "SOUTH 3
MAKE "WEST 4
MAKE "UP 5
MAKE "DOWN 6
MAKE "#DIRNAMES [NORTH EAST SOUTH WEST ►
UP DOWN]
MAKE "#VERBS1 [? LOOK INVENTORY]
MAKE "#VERBS2 [TAKE DROP UNLOCK ►
EXAMINE DRINK]
MAKE "#INVENTORY []
MAKE "#ROOM "FOREST
MAKE "#RESULT []
END

TO INITOBS
PPROP "SWORD "DESCRIPT [A DULL SWORD]
PPROP "CROWBAR "DESCRIPT [A SEARS ►
CROWBAR]
PPROP "ROPE "DESCRIPT [A COIL OF HEMP ►
ROPE]
PPROP "KEY "DESCRIPT [AN IRON KEY]
PPROP "TREES "DESCRIPT [ASH AND BIRCH ►
TREES]
PPROP "TREES "IMMOBILE? "TRUE
PPROP "WINE "DESCRIPT [TASTY WINE]
PPROP "WINE "LIQUID? "TRUE
PPROP "BARREL "DESCRIPT [AN OAKEN ►
BARREL]
PPROP "BARREL "CONTAINS [WINE]
PPROP "BARREL "IMMOBILE? "TRUE
END

```

```

TO INITROOMS
PPROP "ANTE "EXITS [[] GUARD DOOR1 [] ►
[] STAIRS]
PPROP "ANTE "DESCRIPT [IN THE ANTEROOM ►
OF THE FORT]
PPROP "STAIRS "EXITS [[] [] [] ANTE ►
HALL]
PPROP "STAIRS "DESCRIPT [ON SOME ►
STAIRS]
PPROP "LADD "EXITS [[] [] [] HALL [] ►
DOCK]
PPROP "LADD "DESCRIPT [AT THE TOP OF A ►
LADDER]
PPROP "DOOR1 "EXITS [ANTE [] FOREST [] ►
[] []]
PPROP "DOOR1 "DESCRIPT [STOUT IRON ►
DOOR]
PPROP "DOOR1 "KEY "KEY
PPROP "DOOR1 "DOOR? "TRUE
PPROP "DOOR1 "LOCKED? "TRUE
PPROP "DOOR1 "SHUT? "TRUE
PPROP "DOCK "EXITS [[] [] [] LADD ►
[]]
PPROP "DOCK "DESCRIPT [ON A STONE ►
DOCK, BESIDE AN UNDERGROUND ►
STREAM]
PPROP "DOCK "CONTAINS [BARREL]
PPROP "HALL "EXITS [[] LADD [] [] ►
STAIRS []]
PPROP "HALL "DESCRIPT [IN A HIGH, ►
NARROW HALL]
PPROP "HALL "CONTAINS [ROPE]
PPROP "GUARD "EXITS [[] [] [] ANTE [] ►
[]]
PPROP "GUARD "DESCRIPT [IN AN OLD ►
GUARD ROOM]
PPROP "GUARD "CONTAINS [CROWBAR SWORD]
PPROP "CLIFF "EXITS [DOOR1 [] FOREST ►
F1 [] []]
PPROP "CLIFF "DESCRIPT [AT THE BASE OF ►
A CLIFF]
PPROP "CLIFF "CONTAINS [TREES]
PPROP "FOREST "EXITS [CLIFF FOREST ►
FOREST F1 [] []]
PPROP "FOREST "DESCRIPT [IN A FOREST]
PPROP "FOREST "CONTAINS [TREES]
PPROP "F1 "EXITS [CLIFF FOREST F1 F1 ►
[] []]
PPROP "F1 "DESCRIPT [IN A FOREST]
PPROP "F1 "CONTAINS [TREES KEY]
END

```



```

TO REMOVE :THING :LIST
IF EMPTY :LIST [OP []]
IF :THING = FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST REMOVE :THING BF ►
:LIST
END

```

```

TO ITEM :N :LIST
IF :N = 1 [OP FIRST :LIST]
OP ITEM :N - 1 BF :LIST
END

```

```

TO PPROP :NAME :PROP :VALUE
IF NOT NAMEP :NAME [MAKE :NAME []]
MAKE :NAME PPROP1 THING :NAME :PROP ►
:VALUE
END

```

```

TO PPROP1 :PLIST :PROP :VALUE
IF EMPTY :PLIST [OP LIST :PROP ►
:VALUE]

```

```

IF :PROP = FIRST :PLIST [OP FPUT :PROP ►
FPUT :VALUE BF BF :PLIST]
OP FPUT FIRST :PLIST FPUT FIRST BF ►
:PLIST PPROP1 BF BF :PLIST :PROP ►
:VALUE
END

```

```

TO GPROP :NAME :PROP
IF NOT NAMEP :NAME [OP []]
OP GPROP1 :PROP THING :NAME
END

```

```

TO GPROP1 :PROP :PLIST
IF EMPTY :PLIST [OP []]
IF :PROP = FIRST :PLIST [OP FIRST BF ►
:PLIST]
OP GPROP1 :PROP BF BF :PLIST
END

```

```

TO PROPTREE? :OBJ :PROP
OP ( GPROP :OBJ :PROP ) = "TRUE
END

```

Dungeon

Here's what the original author of this game, Jeanry Chandler, has to say about it.

Dungeon has all of the virtues of an adventure game and more. Not only can you play the part of the dauntless adventurer, exploring the dungeons in search of wonder and magic and fabulous treasures. You can also play the part of the all powerful dungeonmaster: laying the traps, preparing the monsters, and, of course, placing the treasure.

When you start DUNGEON, you are given the first level of the dungeon, four rooms of treasures, and traps, monsters, and money. The adventurer who enters this perilous palace can expect to face such horrific monsters as the nimble kobold, the mighty troll, and, worst of all, the fearsome man-eating Logo turtle!

But monsters are not the only inhabitants of the dreaded dungeon. As reward for defeating or avoiding those grumpy gargantuans, that same adventurer can fill his pockets with gold, jewels, magical potions, wands, enchanted swords, armor, and much more.

The successful adventurer who gains enough experience points could find himself going up a level of skill, thereby gaining fighting ability and strength.

After a period of blissful adventure, however, the aspiring adven-

turer will grow weary of the limits of my own possibly limited imagination. And this is where the dungeonmaster and the truly original facet of the game come into play.

The adventurer, or a friend, can become the dungeonmaster and design a new level filled with treasures and trolls. To start out, the dungeonmaster might simply make new rooms. Later he could design new shapes for monsters, new treasures, traps, and tricks for the adventurer to face.

More explanation would be detrimental to your understanding of the program. You must experience it to learn more!

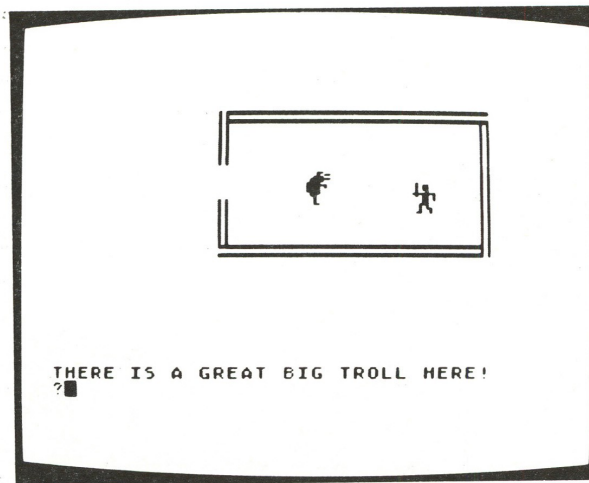
As Jeanry says, there are two ways to enjoy *Dungeon*. One is to be the adventurer in his dungeon, and the other is to add to Jeanry's dungeon or to create your own with his tools. He's right, too, that you must experience it to understand it. So try it!

To start the game, type:

START

Playing the Dungeon Game

START begins the game. You control an adventuring player with a joystick in port 1.* You can move the adventurer with the joystick to avoid or confront monsters, to go through doors, and to get the contents of chests.



As in other adventure games, there are some commands you can type. I stands for inventory, D for drink, and W for wave wand. You must press RETURN after your commands.

*Remember that in Atari Logo, the joystick is referenced by JOY 0 when it is in port 1 of the Atari.

An Overview

The following sections present an overview of how this program works as the game is played and of how to modify the dungeon rooms and create your own dungeon. Then there are some suggestions for modifying and improving this program in more radical ways.

All of the procedures are listed at the end of this write-up. You may want to look at some of them as you read about them.

Rooms

Each room is represented by a single procedure (for example, ROOM1, ROOM2) that prints messages about what is in the room and makes turtles into monsters and treasure chests. Turtle 0 is the player, turtle 1 is usually a monster, and turtle 2 is usually a treasure chest.

MAKEROOM is called as a subprocedure from all rooms. It does stuff that needs to be done for each room: it draws walls and doors, sets variables with the dimensions of the room, and creates some demons. Two of the demons that it creates are those that wait for a collision between the player and the monster and between the player and a treasure chest. It also creates a demon that lets you control the player with the joystick and demons that keep the player and monster from drifting through the walls of the room.

The rest of the instructions in each room procedure customize the room. For example, look at ROOM2 in which turtle 2 is a chest and turtle 1 is a kind of monster called a kobold.

```
TO ROOM2
PR [THERE IS A BIG CHEST HERE]
PR [THERE IS A CUTE KOBOLD HERE]
RT 180
MAKEROOM 200 90 [[N ROOM3] [E ROOM1]]
PU
HOME FD 50
SETSP 20
SET MONSTER 3 [-20 25] 270          Give monster kobold shape.
MAKE "MHITP 2
ASK 1 [SETSP 22]
ASK 2 [PU FD 30 LT 90 FD 25 ST]
END
```

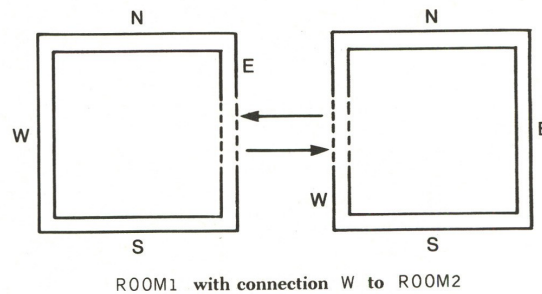
Walls and Doors

The walls and doors are set up for each room by the MAKEROOM procedure. The walls are drawn with pen 0 and the doors with pen 1 and pen 2. A room can have up to two doors.

The SETUP.DEMONS subprocedure of MAKEROOM creates a demon that waits for the player (turtle 0) to bump into a wall; the demon calls a procedure that makes the player bounce back. The condition for this demon is

OVER 0 0. Demons are also created to make the monster bounce off the walls and doors. The monster is not allowed to go through doors.

The DOOR.SIDE subprocedure of MAKEROOM creates demons that wait for the player to bump into a door. These demons use conditions OVER 0 1 or OVER 0 2 to detect this. When the player bumps into a door, the player is moved into the adjoining room. The way this works is that the demon for that door calls the procedure that represents the adjoining room.



The demon for this situation is created by DOOR.SIDE using

```
WHEN OVER 0 1 [ROOM2]
```

The Most Common Actions, FIGHT and CHEST

The SETUP.DEMONS procedure, called by MAKEROOM, creates the demons that wait for the player-monster and player-treasure chest collisions.

The demon that awaits collisions between the player and the monster is created using the instruction WHEN TOUCHING 0 1 [FIGHT]. Turtle 0 is the player and turtle 1 is a monster. When they collide, the demon calls the procedure FIGHT, which causes the two turtles to "fight." They swing at one another. The program considers the strengths and magical aids of the two combatants and determines if either one is hit. As the combatants continue to receive blows, they accumulate "hit points." If either sustains too much damage, it dies.

The demon that awaits collisions between the player and the treasure chest is created using the instruction WHEN TOUCHING 0 2 [CHEST]. Turtle 2 is a treasure chest. If the player collides with turtle 2, the demon calls CHEST. The procedure CHEST determines what treasures the chest contains and rewards the player with those treasures. Usually a treasure is given to the player by changing the value of a global variable such as :GOLD or :ARMOR.

Procedural and Demon-Based Representation

In this program, the only way to figure out all the details is to look at all the procedures. You might say that the program itself "figures it out as it goes along." You might contrast this with Jim Davis's Adventure game. Jim's program has global structures that contain information about his dun-

geon. For example, it has a list of all rooms. In Dungeon, almost all information is in the room procedures.

Programmer as Dungeonmaster: How to Create New and Better Dungeons

In this section I will discuss three kinds of changes to the Dungeon game. You can create new monsters, treasures, and whole rooms to put them in. You can create a new dungeon to replace Jeanry's. You can make changes to the workings of the game program itself to improve and change the game.

Creating New Rooms, Monsters, and Treasures

Looking at the room procedures (ROOM1, ROOM2, and so forth) will help you figure out how to make new rooms. You could make your very own completely new dungeon, or you could add rooms to the existing one. Remember that when you add a room, you might want to change some of the old rooms so that they connect to your new room. You might want to change the W procedure so that the wand can magically teleport the player into your new room.

You may have noticed that Jeanry's doors are *two-way*. That is, if a door leads *west* from ROOM1 to ROOM2, then there is a door that leads *east* from ROOM2 to ROOM1. Jeanry has made all of his doors match up. You might want to make all your doors match up too, or you could make some doors be one-way only. You could make some interesting and confusing dungeons this way.

The MAKEROOM procedure sets :ROOMLENGTH and :ROOMHEIGHT to be the length and height of the room. You can use these to position turtles or drawings in the rooms.

Here's an example of a new room I created.

```

TO ROOM5
PR [THIS IS MARGARET'S ROOM]
PR [BEWARE THE TROLL HERE]
PR [THERE IS A MYSTERIOUS ANCIENT CHEST HERE]
MAKEROOM 100 120 [[W ROOM4] [S ROOM2]]
PU
HOME
SETPOS SE (:ROOMLENGTH / 2) - 50 :ROOMHEIGHT / 2      Player in middle of room.
SETH 45
SETSP 20
SET.MONSTER 2 SE :ROOMLENGTH - 80 :ROOMHEIGHT - 25 300 Troll in corner.
MAKE "MNSTR 1
ASK 1 [SETSP 22]
ASK 2 [PU SETPOS SE :ROOMLENGTH - 90 25 ST]
END

```

This new room has a door leading west to ROOM4 and a door leading south to ROOM2. To make it possible for the player to get to this room, I put a door in ROOM4 leading east to ROOM5. To do this I changed a line in ROOM4 from

GAMES

```
MAKEROOM 100 100 [[S ROOM3]]
```

to

```
MAKEROOM 100 100 [[S ROOM3] [E ROOM5]]
```

I also changed a line in W from

```
IF EQUALP :WAND 2 [PR [YOU ARE TELEPORTED TO A NEW LOCATION]
  RUN FPUT WORD "ROOM 1 + RANDOM 4 []]
```

to

```
IF EQUALP :WAND 2 [PR [YOU ARE TELEPORTED TO A NEW LOCATION]
  RUN FPUT WORD "ROOM 1 + RANDOM 5 []]
```

To add a new kind of monster, you might want to make a new shape for it. You could add it to an old or new room, using the SET.MONSTER procedure. (You will probably want to add instructions about your monster shape in the START and SAVESH procedures.)

You can add new kinds of treasure. You must put your new kind of treasure in the CHEST procedure so it can be "in" the treasure chest. Then you must create a procedure to be run when your new treasure is found. If you want to represent your kind of treasure as a variable with a point value (like GOLD or WAND), you should initialize it in the START procedure.

Making a New Dungeon

Jeanry has left an opening in the program for you to add a complete new dungeon.

When you are playing the game and get to the stairs (in ROOM4), the program asks if you want to go down. If you answer Y, then the program types a message saying that you cannot go down to the lower dungeon unless you create it.

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y
PRINT [YOU CAN ONLY GO TO THE LOWER DUNGEON]
PRINT [IF YOU CREATE IT!]
END
```

Let's say you create several new rooms that connect to each other but do not connect to Jeanry's original four rooms. For example, let's say you make ROOM6, ROOM7, ROOM8, and ROOM9.

Then you could change STAIRS to

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y
ROOM7
END
```


Then STAIRS would plunk the player right into your dungeon.

If you need more Logo workspace for your new dungeon, you could put your new room procedures in a separate file, for example D:LOWERDUNGEON. Then you could have STAIRS erase Jeanry's dungeon and load in yours. For example:

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y
ER [ROOM1 ROOM2 ROOM3 ROOM4]
LOAD "D:LOWERDUNGEON
ROOM7
END
```

Improving and Changing the Dungeon Program

Right now there is only one kind of chest. It can contain any of the kinds of treasure in the game. You could change the game so that there are several kinds of chests with different kinds of treasures in them.

You could make a player who had certain treasures or lots of experience points become more powerful. For example, the player could bribe monsters to go away if he had enough gold. Or the player could be unable to see certain treasure chests unless he found some magic glasses. The Dungeon game could allow the player to go to the lower level only if he had accumulated enough experience points. Here's a way to implement that last suggestion.

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y
IF ((5 * :LEVEL) + :EXPERIENCE) > 18 [GO.DOWN]
  [PRINT [YOU MUST BE WISER TO ENTER THE LOWER DUNGEON]]
END
```

```
TO GO.DOWN
ER [ROOM1 ROOM2 ROOM3 ROOM4]
LOAD "D:LOWERDUNGEON
ROOM7
END
```

You could create more typed commands similar to I, W, and D.

You could make the game smart about what direction you are going when you go through doors.

You could introduce global data structures to keep track of objects. This way the game could know when a monster in a particular room is dead and not display it again when you return to that room.

PROGRAM LISTING

SETTING UP

```

TO START
PUTSH 1 :PLAYER
PUTSH 2 :TROLL
PUTSH 3 :KOBOLD
PUTSH 4 :CHEST
PUTSH 5 :THRUST
PUTSH 6 :STAIRS
MAKE "PHITP 0           :PHITP is hit points against player.
MAKE "MHITP 0           :MHITP is hit points against monster.
MAKE "GOLD 1
MAKE "EXPERIENCE 1
MAKE "SWORD 0
MAKE "MNSTR 0
MAKE "POTION 0
MAKE "WAND 0
MAKE "LEVEL 0
MAKE "ARMOR 0
ASK [0 1 2 3] [PU HOME HT]
TELL 0
ROOM1
END

```

MAKEROOM, THE GENERAL ROOM MAKER

```

TO MAKEROOM :LEN :HGT :DOORS      MAKEROOM assumes WHO is 0.
CS HT
ASK [1 2 3] [HT]
HOME LT 90 FD 50 RT 90
SIDE :HGT :DOORS "W 1
SIDE :LEN :DOORS "N 1
SIDE :HGT :DOORS "E 1
SIDE :LEN :DOORS "S 1
DIMENSIONS :LEN :HGT
SETUP.DEMONS
PU ST
END

TO DIMENSIONS :LEN :HGT
MAKE "ROOMLENGTH :LEN
MAKE "ROOMHEIGHT :HGT
END

```

```

TO SETUP.DEMONS
WHEN OVER 0 0 [BK 10]           For when the player hits solid part of a wall.
WHEN TOUCHING 0 1 [FIGHT]       Turtle 1 is usually a monster.
WHEN TOUCHING 0 2 [CHEST]       Turtle 2 is usually a treasure chest.
WHEN 15 [MOVEPLAYER JOY 0]
WHEN OVER 1 0 [ASK 1 [BK 5 MOVE]]
WHEN OVER 1 1 [ASK 1 [BK 5 MOVE]]
WHEN OVER 1 2 [ASK 1 [BK 5 MOVE]]
END

```

STUFF USED BY MAKEROOM TO DRAW WALLS AND DOORS

```

TO SIDE :LEN :DOORS :WALL :PEN
IF EMPTY :DOORS [SOLID.SIDE :LEN STOP]
IF EQUALP :WALL FIRST FIRST :DOORS ►
  [DOOR.SIDE :LEN FIRST :DOORS :PEN STOP]
SIDE :LEN BF :DOORS :WALL :PEN + 1
END

```

```

TO SOLID.SIDE :LEN
PD
FD :LEN BK :LEN
RT 90
PU
FD 5
LT 90
PD
FD :LEN
RT 90
END

```

```

TO DOOR.SIDE :LEN :DOOR :PEN
PD
DOOR.LINE :LEN :PEN
PU
BK :LEN
RT 90
FD 5
LT 90
PD
DOOR.LINE :LEN :PEN
RT 90
WHEN OVER 0 :PEN BF :DOOR
END

```

```

TO DOOR.LINE :LEN :PEN
FD ( :LEN - 20 ) / 2
SETPN :PEN
FD 20
SETPN 0
FD ( :LEN - 20 ) / 2
END

```

INDIVIDUAL ROOMS

```

TO ROOM1
PR [THERE IS A GREAT BIG TROLL HERE!]
TELL 0
SETSH 1
PD
MAKEROOM 150 80 [[W ROOM2]]
HT
PU HOME FD 30 RT 90 FD 20 ST
SETSP 20

```

```
SET.MONSTER 2 [-35 40] 90
MAKE "MNSTR 2
END
```

Give monster troll shape.

```
TO ROOM2
PR [THERE IS A BIG CHEST HERE]
PR [THERE IS A CUTE KOBOLD HERE]
RT 180
MAKEROOM 200 90 [[N ROOM3] [E ROOM1]]
SETSH 1
HOME FD 50
SETSP 20
SET.MONSTER 3 [-20 25] 270
ASK 1 [SETSP 22]
ASK 2 [PU SETPOS [-25 30] ST]
END
```

Give monster kobold shape.
This is the chest.

```
TO ROOM3
MAKEROOM 20 100 [[N ROOM4] [S ROOM2]]
SETSH 1
SETPOS [-40 30]
SETSP 20
END
```

```
TO ROOM4
PRINT [THERE ARE STAIRS DOWN HERE]
PRINT [THERE IS A FEROCIOUS MAN EATING
      GIANT LOGO TURTLE HERE]
PRINT [THERE IS A GREAT BIG BRONZE BOUND CHEST HERE]
MAKEROOM 100 100 [[S ROOM3]]
SET.MONSTER 0 [-20 50] 180
ASK 2 [ST PU SETSH 4 SETPOS [-30 70]]
ASK 3 [ST PU SETSH 6 SETPOS [20 20]]
WHEN TOUCHING 0 3 [STAIRS]
WHEN OVER 1 0 [ASK 1 [BK 9]]
SETSH 1
SETPOS [-10 30]
SETSP 20
END
```

Give monster turtle shape.
This is the chest.
This is the stairs shape.

ACTIONS THAT HAPPEN IN THE DUNGEON

```
TO MOVEPLAYER :JOY
IF :JOY < 0 [STOP]
SETH 45 * :JOY
ASK 1 [MOVE]
END
```

```
TO FIGHT
SETSH 5
PR [YOU SWING...]
IF EQUALP RANDOM 3 1 [MHIT]
SETSH 1
PR [IT ATTACKS...]
```



```
IF EQUALP RANDOM 3 1 [PHIT :MNSTR]
ASK 1 [BK 10]
BK 15
ASK 1 [MOVE]
END
```

```
TO CHEST
PR [CREAK,]
PR [( THE CHEST OPENS )]
IF EQUALP RANDOM 5 1 [TRAP]
IF EQUALP RANDOM 3 1 [GOLD]
IF EQUALP RANDOM 3 1 [GOLD]
IF EQUALP RANDOM 3 1 [GOLD]
IF EQUALP RANDOM 3 1 [GOLD]
IF EQUALP RANDOM 3 1 [GOLD]
IF EQUALP RANDOM 2 1 [POTION]
IF EQUALP RANDOM 3 1 [JEWEL]
IF EQUALP RANDOM 3 1 [SWORD]
IF EQUALP RANDOM 3 1 [WAND]
IF EQUALP RANDOM 2 1 [ARMOR]
ASK 2 [HT]
END
```

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
IF NOT KEY "Y [BK 5 STOP]
PRINT [YOU CAN ONLY GO TO THE LOWER DUNGEON]
PRINT [IF YOU CREATE IT!]
ASK 3 [HT]
END
```

```
TO KEY :K
OP EQUALP RC :K
END
```

ACTIONS THAT HAPPEN BECAUSE OF GETTING TREASURES

```
TO TRAP
PR [YOU HIT A TRAP!]
PHIT 1
END
```

```
TO GOLD
PRINT "GOLD!
ADD "GOLD 5
ADD "EXPERIENCE 1
IF :EXPERIENCE > 20 [LEVEL]
END
```

```
TO POTION
PR [A POTION!]
MAKE "POTION 1 + RANDOM 4
END
```

```

TO JEWEL
PRINT "JEWEL!!
ADD "GOLD 50
ADD "EXPERIENCE 5
IF :EXPERIENCE > 20 [LEVEL]
END

```

```

TO SWORD
PR [A FINE ELFBLADE,WORTHY OF YOUR SKILL]
ADD "SWORD RANDOM 4
END

```

```

TO WAND
MAKE "WAND 1 + RANDOM 3
END

```

```

TO ARMOR
MAKE "ARMOR 1 + RANDOM 4
END

```

```

TO LEVEL
ADD "LEVEL 1
PR (SE [YOU NOW HAVE] :LEVEL [LEVELS OF EXPERIENCE])
MAKE "EXPERIENCE 0
END

```

FIGHTING

```

TO MHIT
PRINT "CRUNCH!
ADD "MHITP 1 + RANDOM 3
ADD "MHITP :SWORD
IF :MHITP > 5 [MDEAD]
ASK 1 [SETSP SUM SPEED 1]
END

```

Monster is mad after hit, gets faster.

```

TO MDEAD
PR [YOU KILLED THE MONSTER]
ADD "EXPERIENCE 5 + :MNSTR
IF :EXPERIENCE > 20 [LEVEL]
MAKE "MHITP 0
ASK 1 [SETSH 6 TOOT 0 255 5 20 HT ]
END

```

```

TO PHIT :STRENGTH
PRINT "OUCH
ADD "PHITP 1 + RANDOM 2
ADD "PHITP :STRENGTH
IF :PHITP > 10 [PDEAD]
END

```

```

TO PDEAD
PR [THOU ART SLAIN!]

```

```
T00T 0 255 1 10
ASK [0 1 2 3] [CS HT]
END
```

I FOR INVENTORY

```
TO I
TYPE "GOLD PR :GOLD
TYPE "LEVEL PR :LEVEL
TYPE "EXPERIENCE PR :EXPERIENCE
TYPE [HIT POINTS] PR :PHITP
IF :SWORD > 0 [PR "ELFBLADE]
IF :WAND > 0 [PR "WAND]
IF :POTION > 0 [PR "POTION]
IF :ARMOR > 0 [PR [MAGICAL CHAINMAIL]]
PR [LEATHER JERKIN]
PR [SACK]
PR [MACE]
END
```

D FOR DRINK

```
TO D
IF EQUALP :POTION 0 [PR [YOU HAVE NO POTION TO DRINK] STOP]
PR [YOU DRINK THE LIQUID]
IF EQUALP :POTION 1 [HPOTION MAKE "POTION 0]
IF EQUALP :POTION 2 [SPOTION MAKE "POTION 0]
IF EQUALP :POTION 3 [HPOTION MAKE "POTION 0]
IF EQUALP :POTION 4 [PPOTION]
END
```

POTIONS AND WHAT THEY DO

```
TO PPOTION
PR [YUCK! YOUR STOMACH WILL NEVER FORGIVE YOU ►
  FOR PUTTING THIS FOUL EXCREMENT IN IT]
PHIT 0
END
```

```
TO SPOTION
ADD "STR 1
PR [YOU QUAFF THE FLUID AND YOUR MUSCLES BULGE!]
END
```

```
TO HPOTION
ADD "H -3
PRINT [YOU FEEL HEALED, TIS A FINE ELIXIR YOU HAVE QUAFFED!]
END
```


W FOR WAVE WAND

```

TO W
IF EQUALP :WAND 0 [STOP]
IF EQUALP :WAND 1 [PR [A BOLT OF LIGHTNING ►
    STREAKS FROM YOUR HAND...] MHIT]
IF EQUALP :WAND 2 [PR [YOU ARE TELEPORTED TO A NEW ►
    LOCATION] RUN FPUT WORD "ROOM 1 + RANDOM 4 []]
IF EQUALP :WAND 3 [PR [THE WAND EXPLODES IN YOUR HAND!] ►
    PHIT 2]
END

```

SETTING UP PARTICULAR MONSTERS

```

TO SET.MONSTER :SHAPE :POS :HEADING
ASK 1 [SETSP 10 PU SETSH :SHAPE ►
    SETPOS :POS SETH :HEADING ST]
END

```

AIMING THE MONSTER TOWARD THE PLAYER

This procedure points the monster toward the quadrant the player is in.

```

TO MOVE
MAKE "XPLAYER ASK 0 [XCOR]
MAKE "YPLAYER ASK 0 [YCOR]
TELL 1
SETH 45
IF :YPLAYER < YCOR [SETH 135]
IF :XPLAYER < XCOR [IF :YPLAYER < YCOR ►
    [SETH 225] [SETH 315]]
TELL 0
END

```

UTILITIES

```

TO ADD :VAR :NUM
MAKE :VAR :NUM + THING :VAR
END

```

```

TO SAVESH
MAKE "PLAYER GETSH 1
MAKE "TROLL GETSH 2
MAKE "KOBOLD GETSH 3
MAKE "CHEST GETSH 4
MAKE "THRUST GETSH 5
MAKE "STAIRS GETSH 6
END

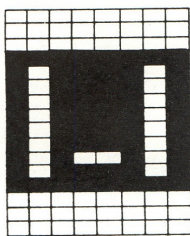
```

```

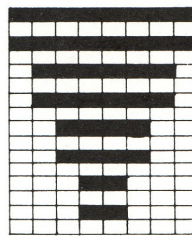
MAKE "THRUST [0 0 12 13 13 37 255 44
              12 12 12 12 60 36 39 101]
MAKE "TROLL [12 11 28 63 60 120 126 122
              120 120 56 48 16 16 16 24]
MAKE "PLAYER [0 64 88 88 88 72 254 90
              26 26 24 56 40 44 36 100]
MAKE "KOBOLD [128 128 152 184 184 136 152 248
              152 24 24 8 8 8 8 24]
MAKE "STAIRS [255 0 255 0 126 0 126 0 60 0 60 0 24 0 24 0]
MAKE "CHEST [0 0 0 255 189 189 189 189
              189 189 165 189 255 0 0 0]

```

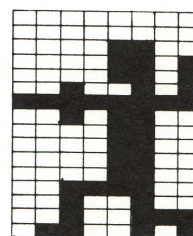
SHAPES



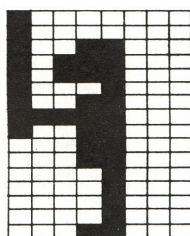
:CHEST
slot 4



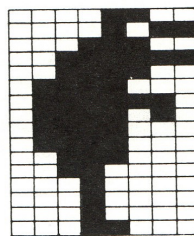
:STAIRS
slot 6



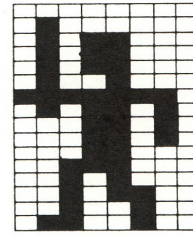
:THRUST
slot 5



:KOBOLD
slot 3



:TROLL
slot 2



:PLAYER
slot 1